US005270921A

United States Patent [19]

Hornick

[11] Patent Number: 5,270,921

[45] Date of Patent: Dec. 14, 1993

[54] **VIRTUAL FARE METHODS FOR A COMPUTERIZED AIRLINE SEAT INVENTORY CONTROL SYSTEM**

[75] Inventor: Scot W. Hornick, Naperville, Ill.

[73] Assignee: Andersen Consulting, Chicago, Ill.

[21] Appl. No.: 640,077

[22] Filed: Jan. 11, 1991

Related U.S. Application Data

[63] Continuation-in-part of Ser. No. 630,261, Dec. 19, 1990.

[51] Int. Cl.[5] G06F 15/26

[52] U.S. Cl. 364/407; 364/401; 364/403

[58] Field of Search 364/407, 401, 403

[56] **References Cited**

U.S. PATENT DOCUMENTS

2,542,890	2/1951	Basu et al.	340/825.37
2,883,106	4/1959	Cornwell et al.	235/385
4,862,357	8/1989	Ahlstrom et al.	364/407

OTHER PUBLICATIONS

W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, "Numerical Recipes," Cambridge University Press, 1986, pp. 300-307.

D. G. Luenberger, "Introduction to Linear and Nonlinear Programming," Addison-Wesley Publishing Company, 1973, pp. 168-188.

S. L. Brumele, J. I. McGill, T. H. Oum, K. Sawaki and M. W. Tretheway "Allocation of Airline Seats between Stochastically Dependent Demands," Trans. Science, vol. 24, No. 3, Aug. 1990.

R. W. Simpson, "Using Network Flow Techniques to Find Shadow Prices for Market Demands and Seat Inventory Control", Flight Trans. Lab Memorandum M89-1, Jan. 1989.

E. L. Williamson and P. P. Belobaba, "Optimization Techniques for Seat Inventory Control," Proceedings of the 28th Annual AGIFORS Symposium, 1988.

A. Jessop, "Optimal Seat Allocation for Airline Planning," 5th International Symposium on Forecasting, 1985.

R. E. Curry, "Optimal Airline Seat Allocation with Fare Classes Nested by Origins and Destinations," Trans. Science, vol. 24, #3 Aug. 1990.

P. P. Belobaba, "Application f a probabilistic decision model to airline seat inventory control," 37 Oper. Rsrch No. 2, Mar. 1989.

F. Glover, R. Glover, J. Lorenzo, and C. McMillan, "The passenger-mix problem in the scheduled airlines," 12 Interfaces #3, Jun. 1982.

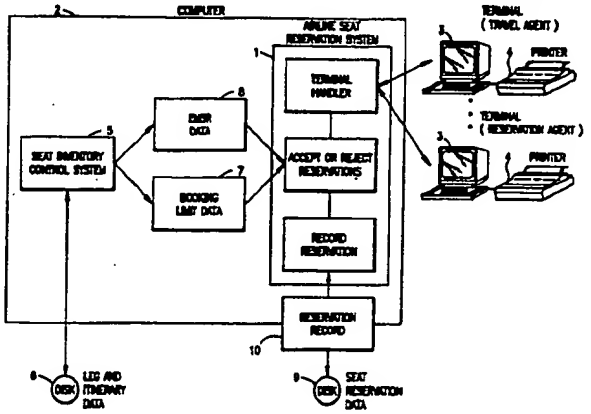
(List continued on next page.)

Primary Examiner—Roy N. Envall, Jr.
Assistant Examiner—Gita Shingala
Attorney, Agent, or Firm—Merchang, Gould, Smith, Edell, Welter & Schmidt

[57] **ABSTRACT**

An airline seat reservation system wherein seat reservations are controlled using, in part, a computerized seat inventory control system. The seat inventory control system, based on a concept termed Network-Based Expected Marginal Seat Revenue (EMSR), does not require the large number of variables required by the other network-based approaches, and it incorporates a probabilistic demand model without resorting to computationally intractable integer programming. The seat inventory control system uses iterative leg-based methods to control bookings in a flight network comprised of a plurality of itinerary/fare class combinations using a plurality of flight legs. When considering a particular flight leg, the total fare paid by a passenger using the leg is adjusted by taking into account an estimate of the displacement cost of the travel on the other legs of the itinerary to create a virtual fare. Expected marginal seat revenues (or more precisely, their current approximations) provide the displacement costs on the legs when computing virtual fares. Using these virtual fares, a leg-based optimization method is applied to the individual legs one-by-one to compute new approximations of the expected marginal seat revenues. This method is iterated until the expected marginal seat revenues concerge to their network-optimal values.

22 Claims, 27 Drawing Sheets



OTHER PUBLICATIONS

R. D. Wollmer, "An airline reservation model for opening and closing fare classes," Unpublished Internal Rpt., McDonnell-Douglas Corp. 1985.

M. Mayer, "Seat allocation, or a simple model of seat allocation via sophisticated ones," 16th Annual Symposium of AGIFORS, Sep. 1976.

S. Dembo & T. Steihaug, "Truncated-Newton algorithms for large scale unconstrained optimization," Mathematical Prog. 26, 1983.

D. P. Adhlfeld, J. M. Mulvey, R. S. Dembo, and S. A. Zenios, "Nonlinear programming on generalized networks," 13 ACM Transactions on Mathematical Software No. 4, 1987.

K. Littlewood, "Forecasting and control of passenger bookings," Proceedings of the 12th AGIFORS Symposium, Oct. 1972, pp. 95-117.

H. Richter, "The differential revenue method to determine optimal seat allotments by fare type," Proceedings of the 22nd AGIFORS Symposium Oct. 1982.

P. P. Belobaba, "Air travel demand and airline seat inventory management," Technical Rpt FTL-R87-8, Flight Trans. Lab. Mass. Inst. of Technology May 1987.

P. P. Belobaba, "Airline yield management, an overview of seat inventory control," 21 Transportation Science No. 2, May 1987 pp. 63-72.

E. L. Williamson, "Comparison of the optimization techniques for origin-destination seat inventory control", Tech. Rpt. FTL-R88-2, Flight Transportation Lab., Massachusetts Institute of Technology, Cambridge, Mass., May 1988.

E. L. Williamson, "Revenue Impacts of Seat Inventory Control," presented to the Third International Airline Yield Management Conference, London, England, Dec. 3, 1990.

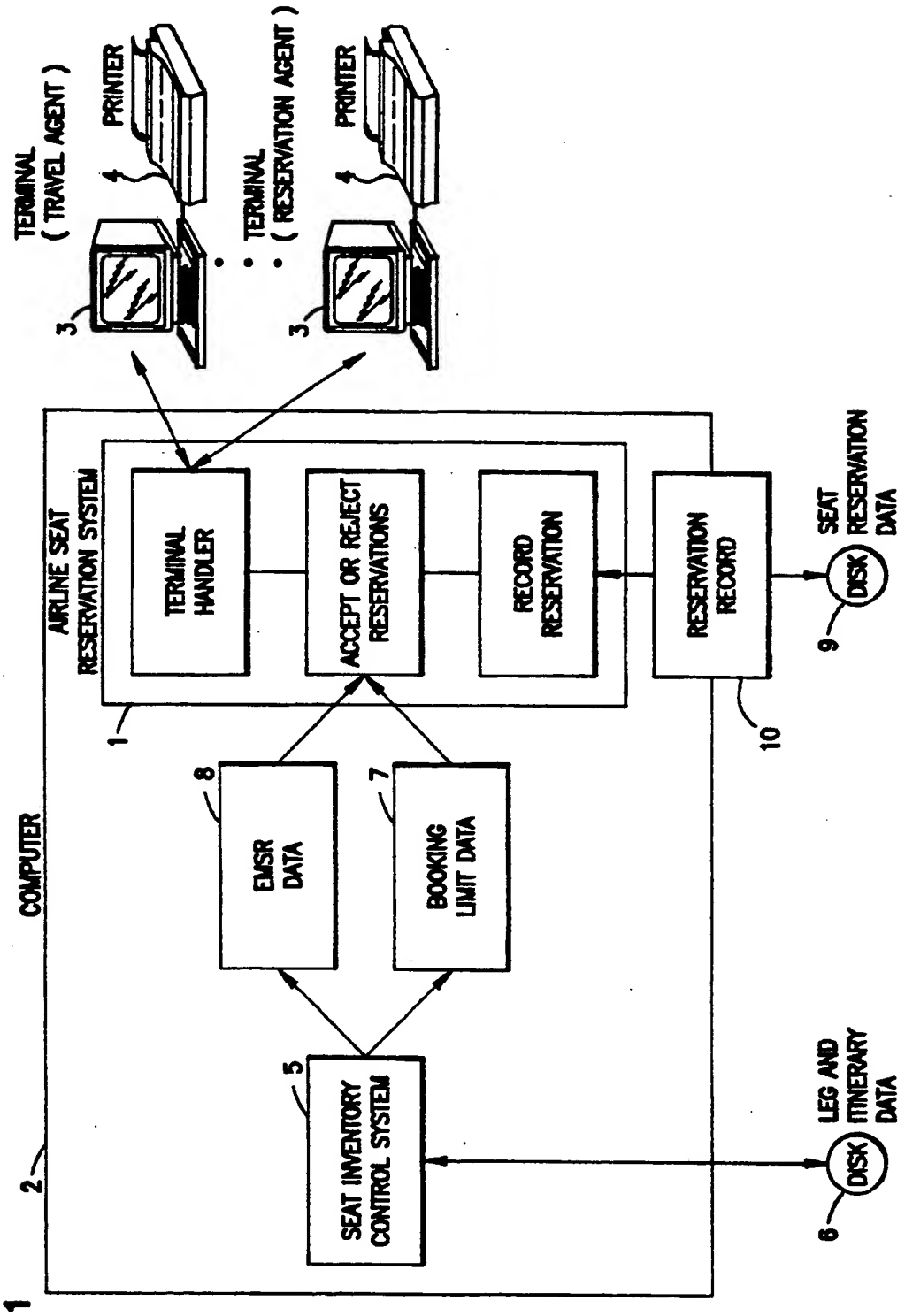

TERMINAL (TRAVEL AGENT)
PRINTER
4
3
TERMINAL (RESERVATION AGENT)
PRINTER
4
3
AIRLINE SEAT RESERVATION SYSTEM
TERMINAL HANDLER
ACCEPT OR REJECT RESERVATIONS
RECORD RESERVATION
RESERVATION RECORD
SEAT RESERVATION DATA
DISK
9
1
10
COMPUTER
8
EMSR DATA
7
BOOKING LIMIT DATA
5
SEAT INVENTORY CONTROL SYSTEM
2
LEG AND ITINERARY DATA
DISK
6

FIG. 1

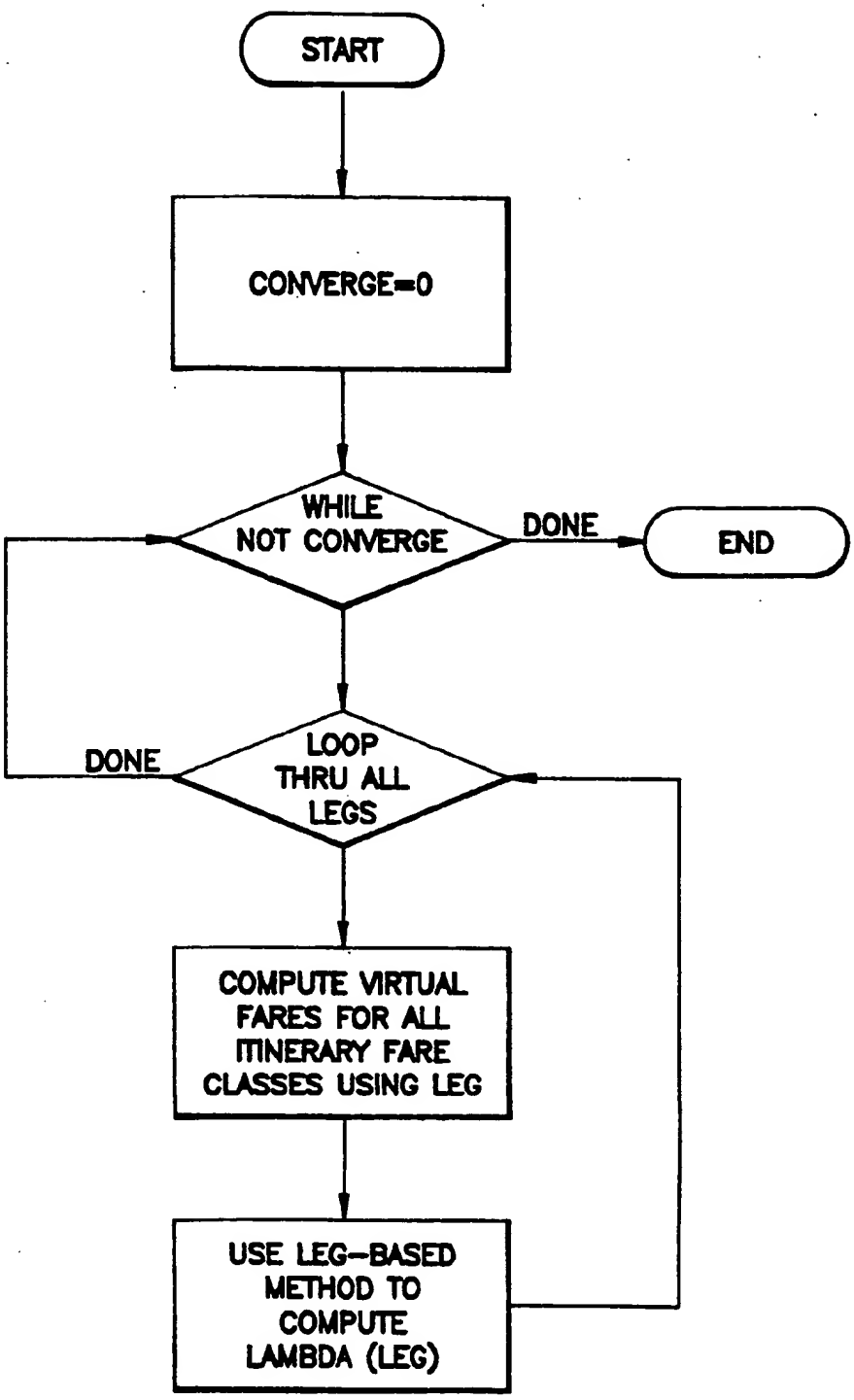

START
CONVERGE=0
WHILE NOT CONVERGE
DONE
END
DONE
LOOP THRU ALL LEGS
COMPUTE VIRTUAL FARES FOR ALL ITINERARY FARE CLASSES USING LEG
USE LEG-BASED METHOD TO COMPUTE LAMBDA (LEG)

FIG. 2

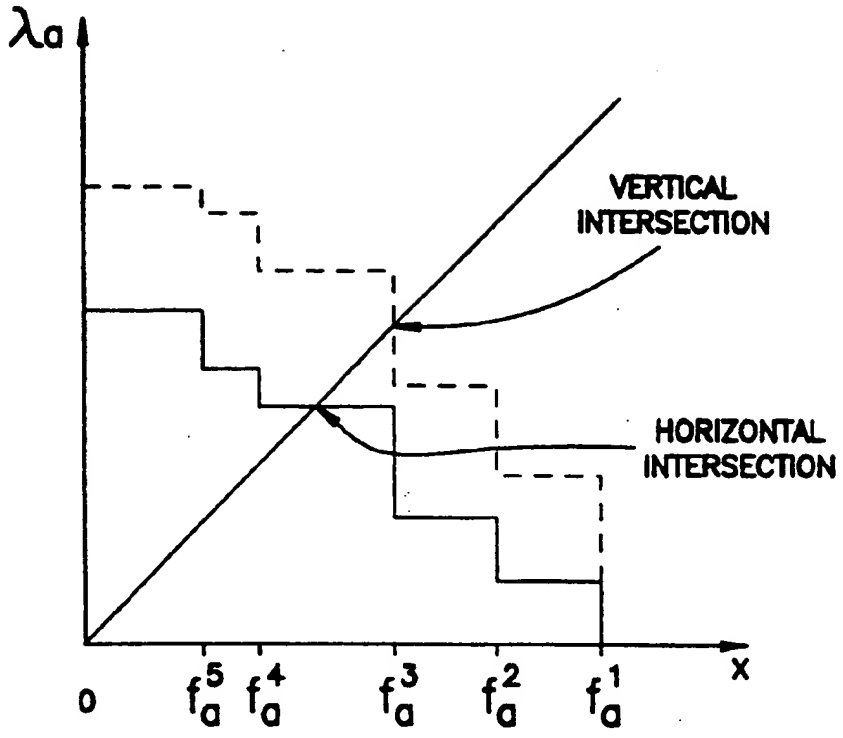

λa
VERTICAL
INTERSECTION
HORIZONTAL
INTERSECTION
0
f_a^5
f_a^4
f_a^3
f_a^2
f_a^1
X

FIG. 3

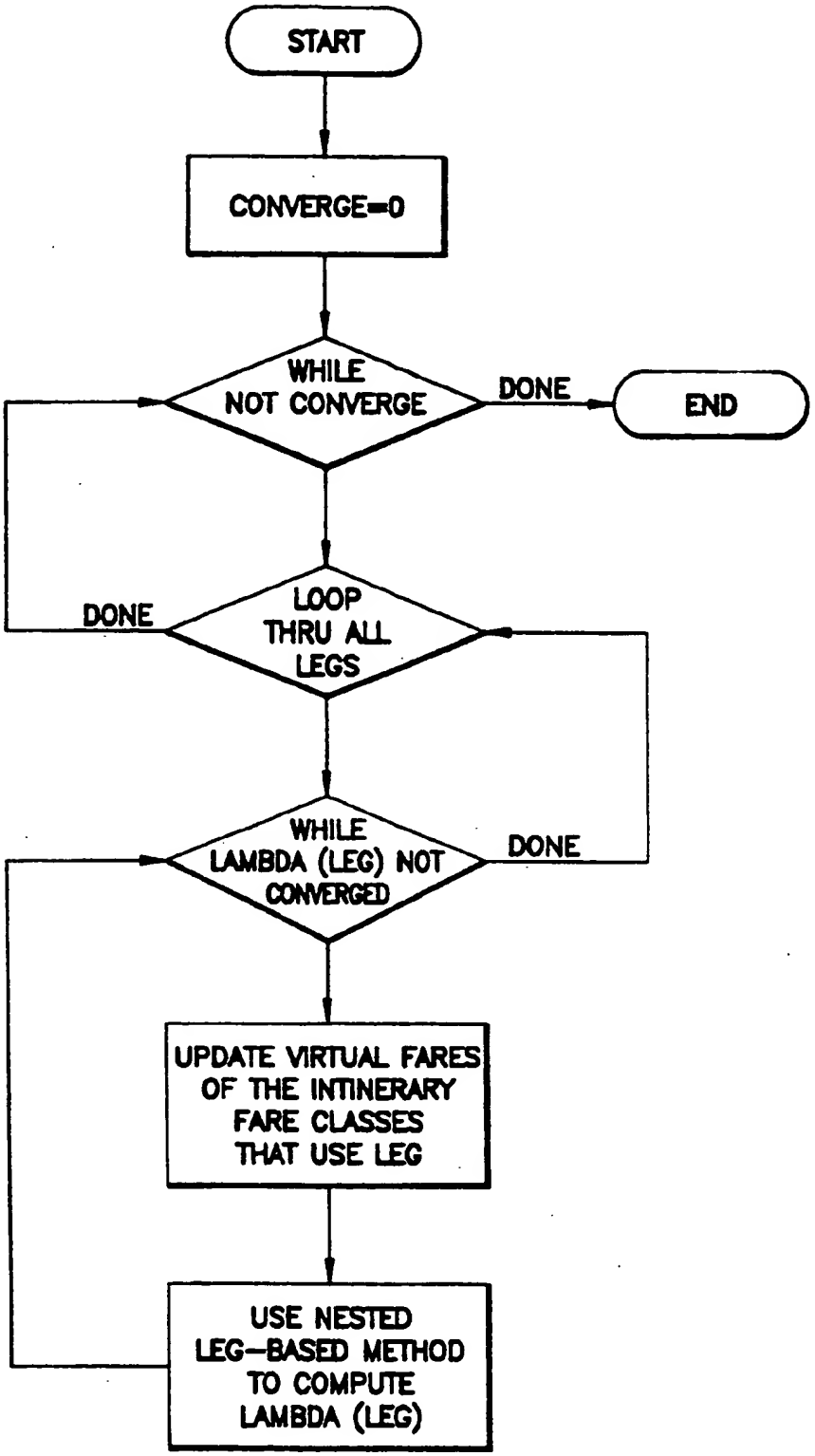

START
CONVERGE=0
WHILE NOT CONVERGE
DONE
END
DONE
LOOP THRU ALL LEGS
WHILE LAMBDA (LEG) NOT CONVERGED
DONE
UPDATE VIRTUAL FARES OF THE INTINERARY FARE CLASSES THAT USE LEG
USE NESTED LEG-BASED METHOD TO COMPUTE LAMBDA (LEG)

FIG. 4

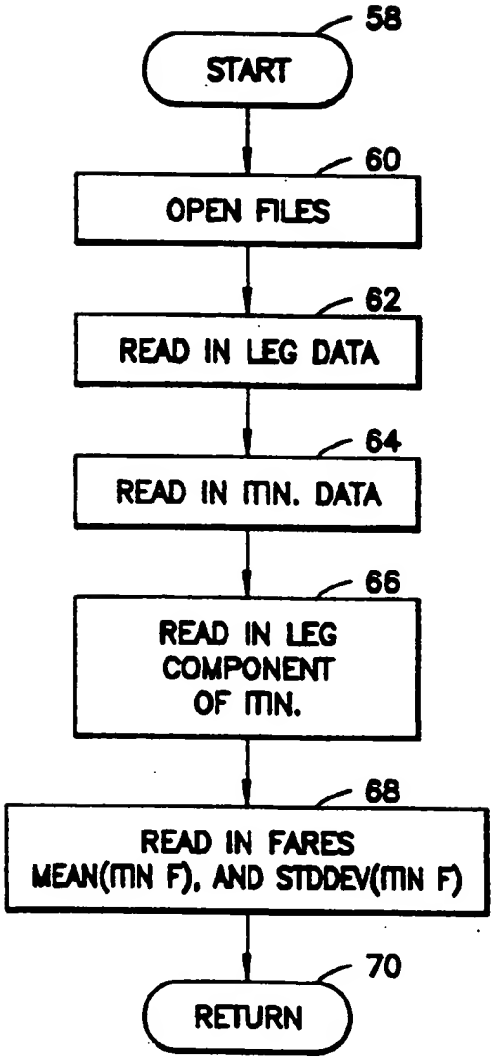
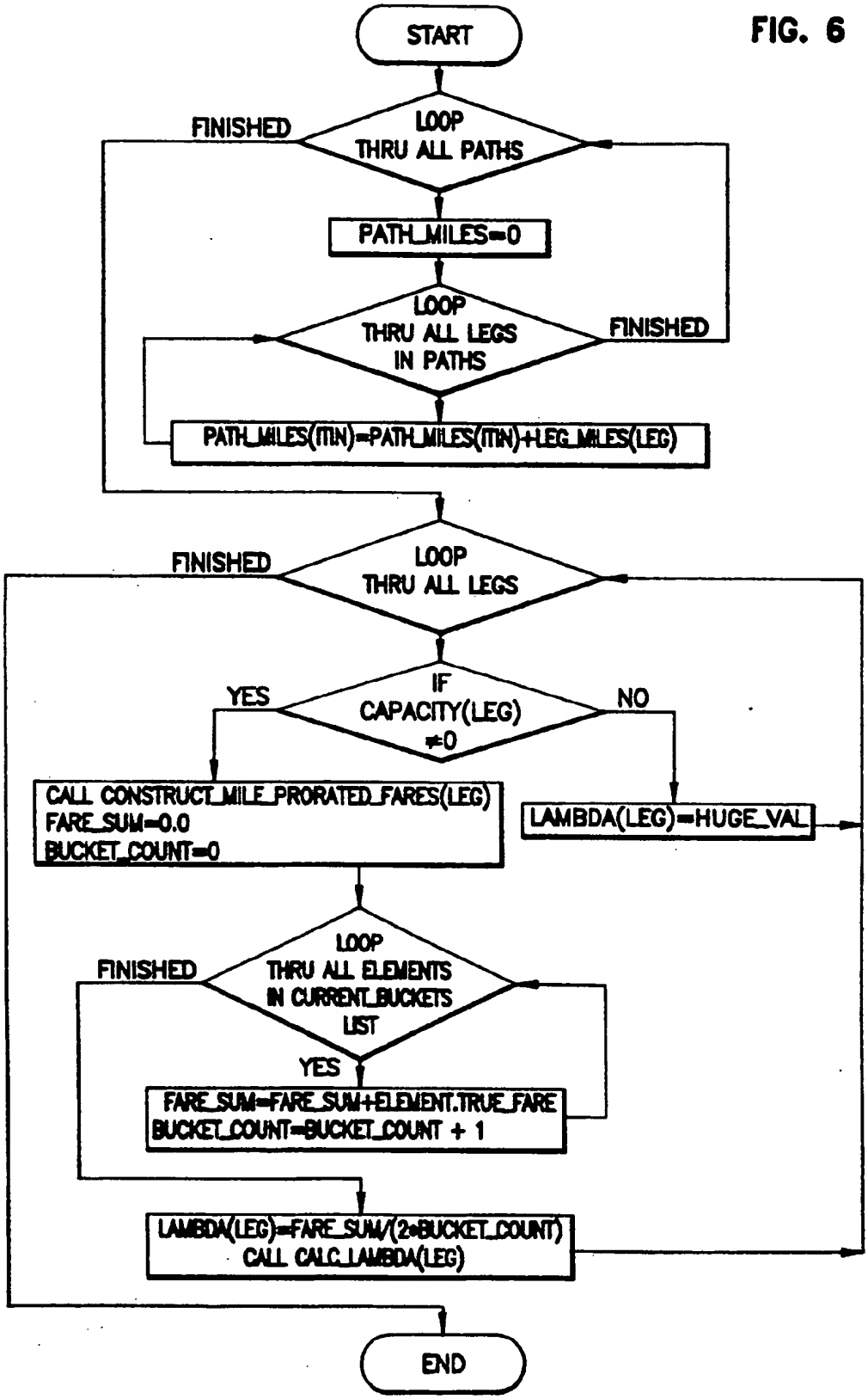

58
START
60
OPEN FILES
62
READ IN LEG DATA
64
READ IN ITIN. DATA
66
READ IN LEG COMPONENT OF ITIN.
68
READ IN FARES MEAN(ITIN F), AND STDDEV(ITIN F)
70
RETURN

FIG. 5

FIG. 6


START
LOOP THRU ALL PATHS
FINISHED
PATH_MILES=0
LOOP THRU ALL LEGS IN PATHS
FINISHED
PATH_MILES(ITIN)=PATH_MILES(ITIN)+LEG_MILES(LEG)
LOOP THRU ALL LEGS
FINISHED
IF CAPACITY(LEG) ≠0
YES
NO
CALL CONSTRUCT_MILE_PRORATED_FARES(LEG)
FARE_SUM=0.0
BUCKET_COUNT=0
LAMBDA(LEG)=HUGE_VAL
LOOP THRU ALL ELEMENTS IN CURRENT_BUCKETS LIST
FINISHED
YES
FARE_SUM=FARE_SUM+ELEMENT.TRUE_FARE
BUCKET_COUNT=BUCKET_COUNT + 1
LAMBDA(LEG)=FARE_SUM/(2*BUCKET_COUNT)
CALL CALC_LAMBDA(LEG)
END

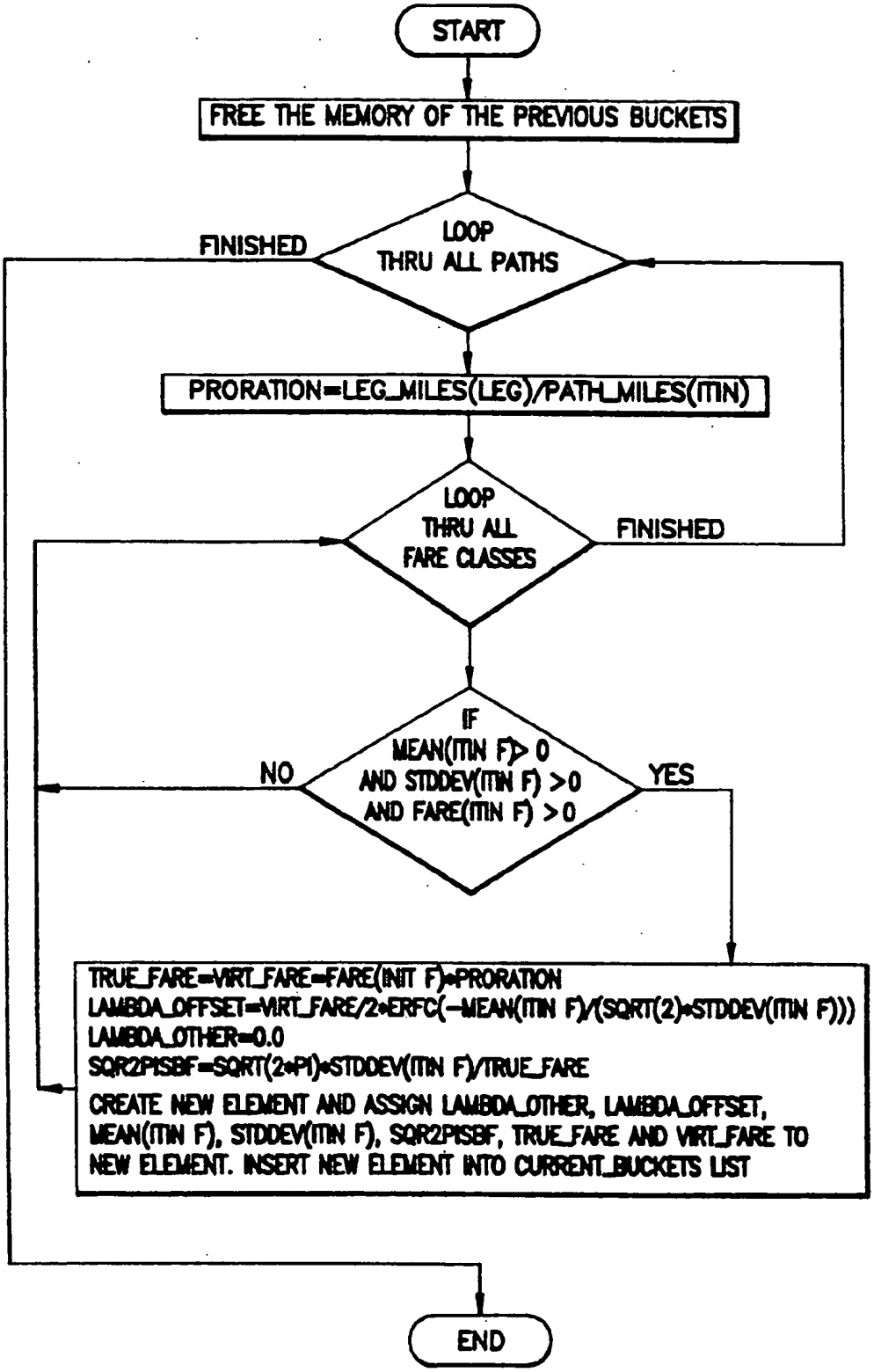

START
FREE THE MEMORY OF THE PREVIOUS BUCKETS
LOOP THRU ALL PATHS
FINISHED
PRORATION=LEG_MILES(LEG)/PATH_MILES(ITIN)
LOOP THRU ALL FARE CLASSES
FINISHED
IF MEAN(ITIN F)> 0 AND STDDEV(ITIN F) >0 AND FARE(ITIN F) >0
NO
YES
TRUE_FARE=VIRT_FARE=FARE(INIT F)*PRORATION
LAMBDA_OFFSET=VIRT_FARE/2*ERFC(-MEAN(ITIN F)/(SQRT(2)*STDDEV(ITIN F)))
LAMBDA_OTHER=0.0
SQR2PISBF=SQRT(2*PI)*STDDEV(ITIN F)/TRUE_FARE
CREATE NEW ELEMENT AND ASSIGN LAMBDA_OTHER, LAMBDA_OFFSET, MEAN(ITIN F), STDDEV(ITIN F), SQR2PISBF, TRUE_FARE AND VIRT_FARE TO NEW ELEMENT. INSERT NEW ELEMENT INTO CURRENT_BUCKETS LIST
END

FIG. 7

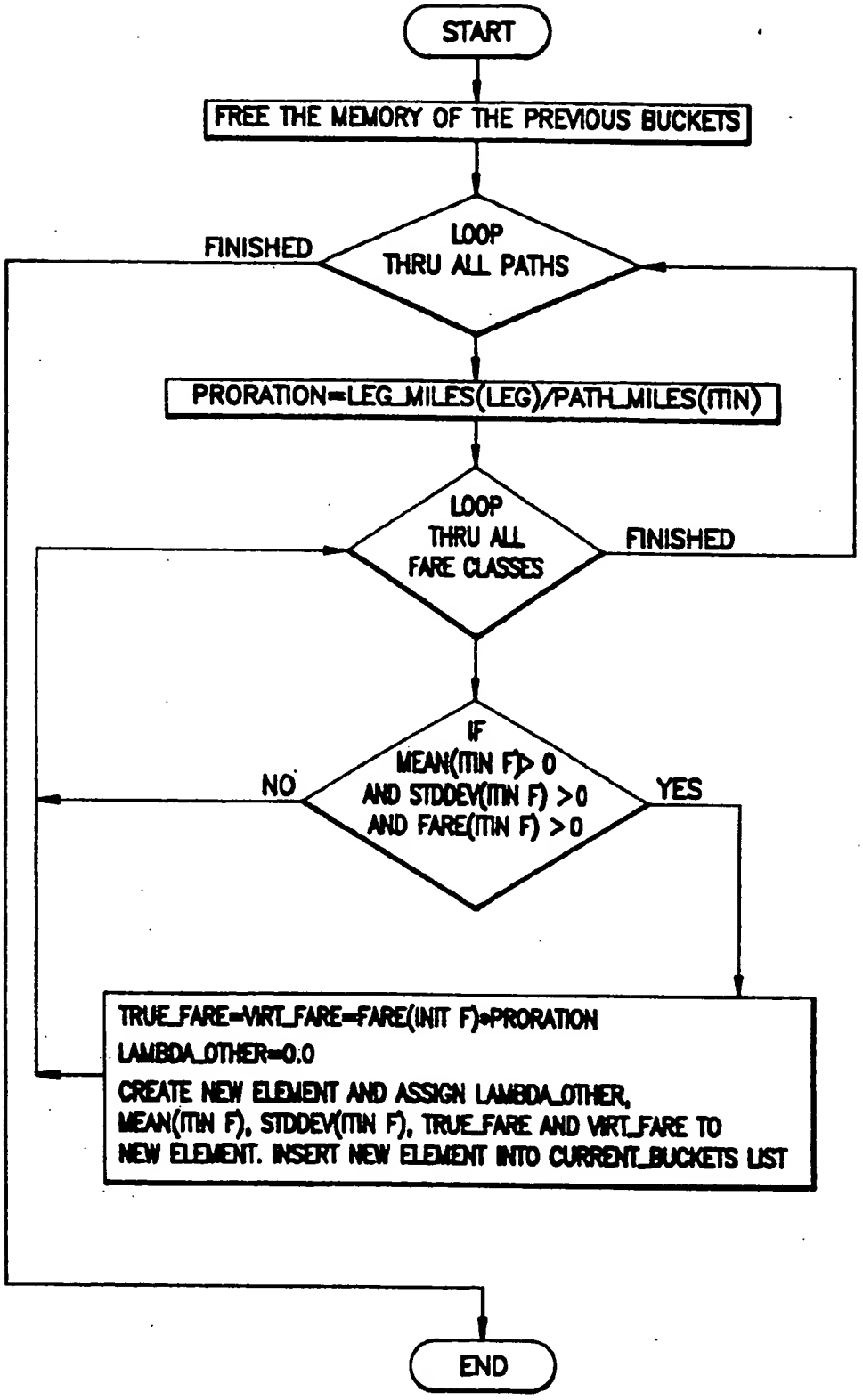

START
FREE THE MEMORY OF THE PREVIOUS BUCKETS
LOOP THRU ALL PATHS
FINISHED
PRORATION=LEG_MILES(LEG)/PATH_MILES(ITIN)
LOOP THRU ALL FARE CLASSES
FINISHED
IF MEAN(ITIN F)> 0 AND STDDEV(ITIN F) >0 AND FARE(ITIN F) >0
NO
YES
TRUE_FARE=VIRT_FARE=FARE(INIT F)*PRORATION
LAMBDA_OTHER=0.0
CREATE NEW ELEMENT AND ASSIGN LAMBDA_OTHER, MEAN(ITIN F), STDDEV(ITIN F), TRUE_FARE AND VIRT_FARE TO NEW ELEMENT. INSERT NEW ELEMENT INTO CURRENT_BUCKETS LIST
END

FIG. 8

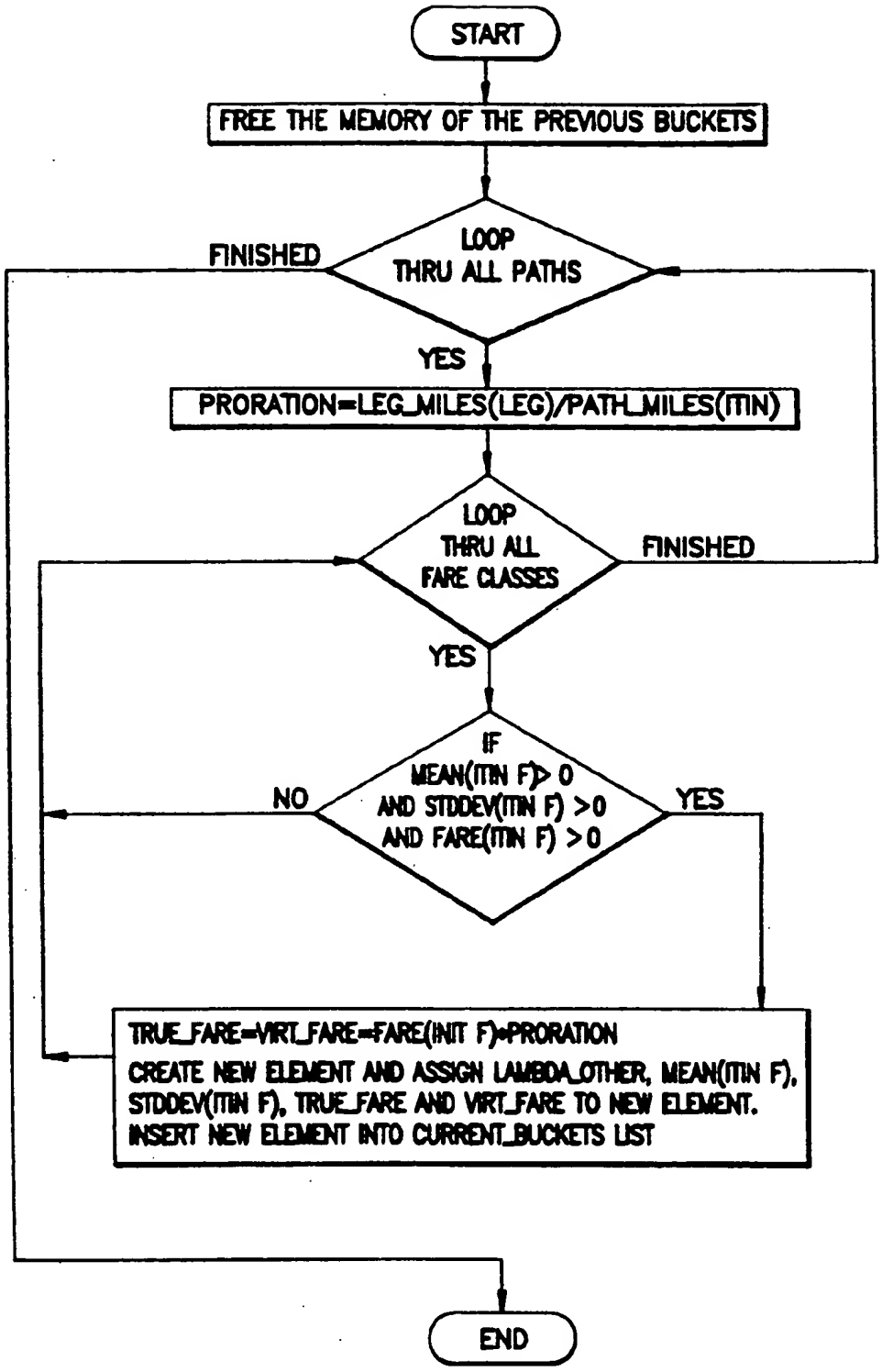

START
FREE THE MEMORY OF THE PREVIOUS BUCKETS
LOOP THRU ALL PATHS
FINISHED
YES
PRORATION=LEG_MILES(LEG)/PATH_MILES(ITIN)
LOOP THRU ALL FARE CLASSES
FINISHED
YES
IF MEAN(ITIN F)> 0 AND STDDEV(ITIN F) >0 AND FARE(ITIN F) >0
NO
YES
TRUE_FARE=VIRT_FARE=FARE(INIT F)*PRORATION
CREATE NEW ELEMENT AND ASSIGN LAMBDA_OTHER, MEAN(ITIN F), STDDEV(ITIN F), TRUE_FARE AND VIRT_FARE TO NEW ELEMENT. INSERT NEW ELEMENT INTO CURRENT_BUCKETS LIST
END

FIG. 9

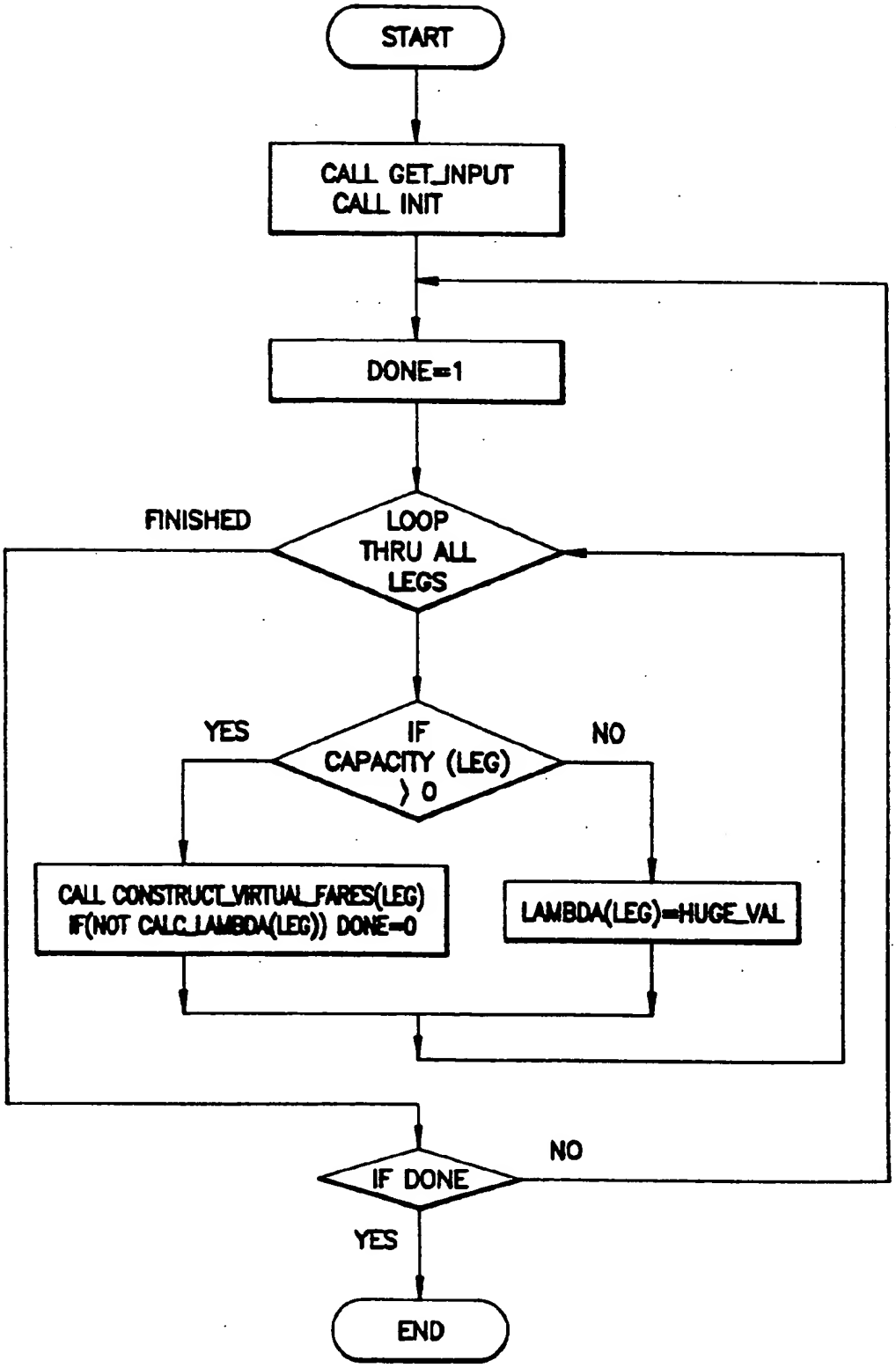

START
CALL GET_INPUT
CALL INIT
DONE=1
FINISHED
LOOP
THRU ALL
LEGS
YES
IF
CAPACITY (LEG)
> 0
NO
CALL CONSTRUCT_VIRTUAL_FARES(LEG)
IF(NOT CALC_LAMBDA(LEG)) DONE=0
LAMBDA(LEG)=HUGE_VAL
NO
IF DONE
YES
END

FIG. 10

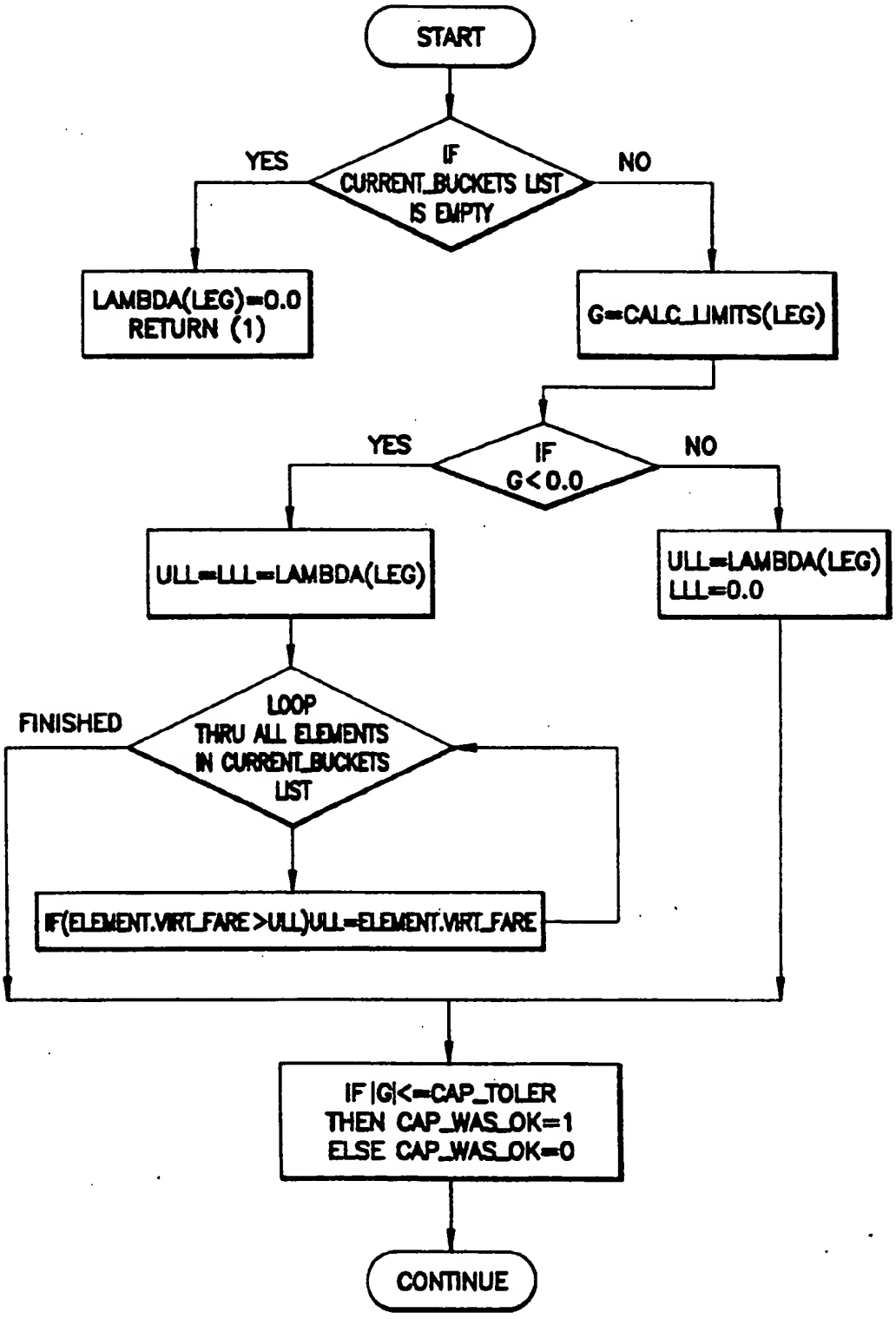

START
IF CURRENT_BUCKETS LIST IS EMPTY
YES
NO
LAMBDA(LEG)=0.0 RETURN (1)
G=CALC_LIMITS(LEG)
IF G<0.0
YES
NO
ULL=LLL=LAMBDA(LEG)
ULL=LAMBDA(LEG) LLL=0.0
LOOP THRU ALL ELEMENTS IN CURRENT_BUCKETS LIST
FINISHED
IF(ELEMENT.VIRT_FARE>ULL)ULL=ELEMENT.VIRT_FARE
IF |G|<=CAP_TOLER THEN CAP_WAS_OK=1 ELSE CAP_WAS_OK=0
CONTINUE

FIG. 11A

FIG. 11B

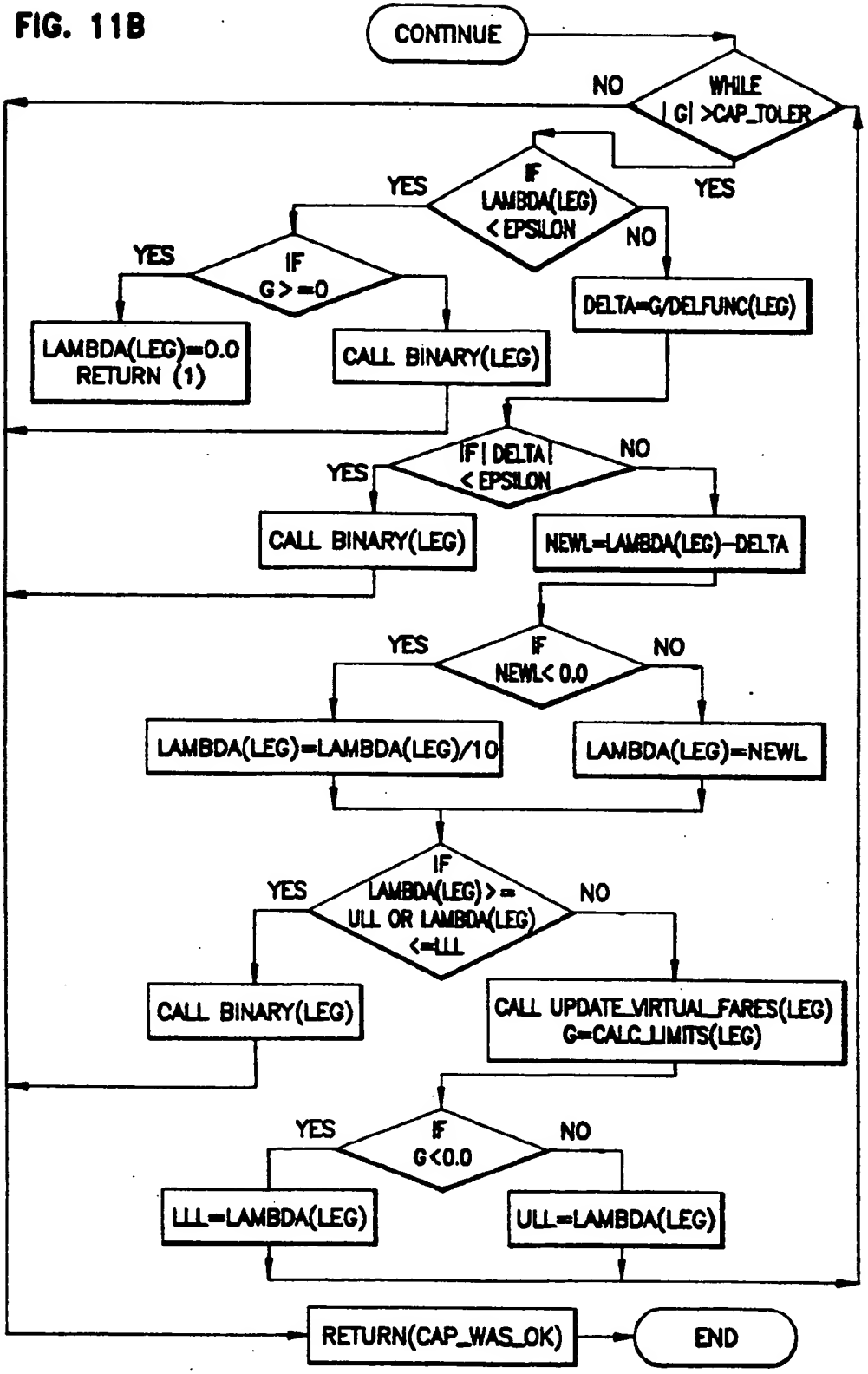

CONTINUE
WHILE |G| >CAP_TOLER
NO
YES
IF LAMBDA(LEG) < EPSILON
YES
NO
IF G>=0
YES
LAMBDA(LEG)=0.0 RETURN (1)
CALL BINARY(LEG)
DELTA=G/DELFUNC(LEG)
IF |DELTA| < EPSILON
YES
NO
CALL BINARY(LEG)
NEWL=LAMBDA(LEG)-DELTA
IF NEWL< 0.0
YES
NO
LAMBDA(LEG)=LAMBDA(LEG)/10
LAMBDA(LEG)=NEWL
IF LAMBDA(LEG)>= ULL OR LAMBDA(LEG) <=LLL
YES
NO
CALL BINARY(LEG)
CALL UPDATE_VIRTUAL_FARES(LEG) G=CALC_LIMITS(LEG)
IF G<0.0
YES
NO
LLL=LAMBDA(LEG)
ULL=LAMBDA(LEG)
RETURN(CAP_WAS_OK)
END

FIG. 12A

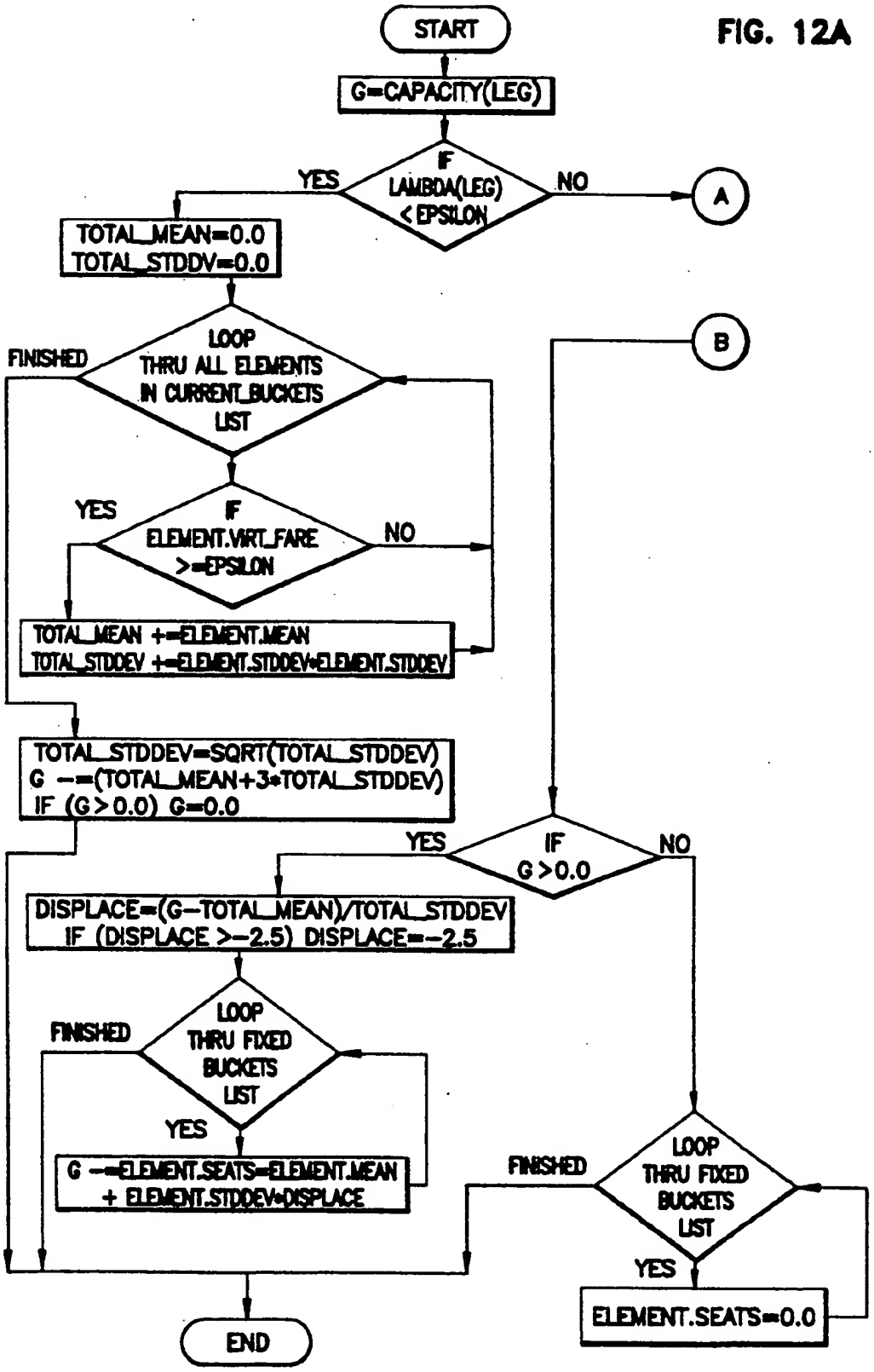
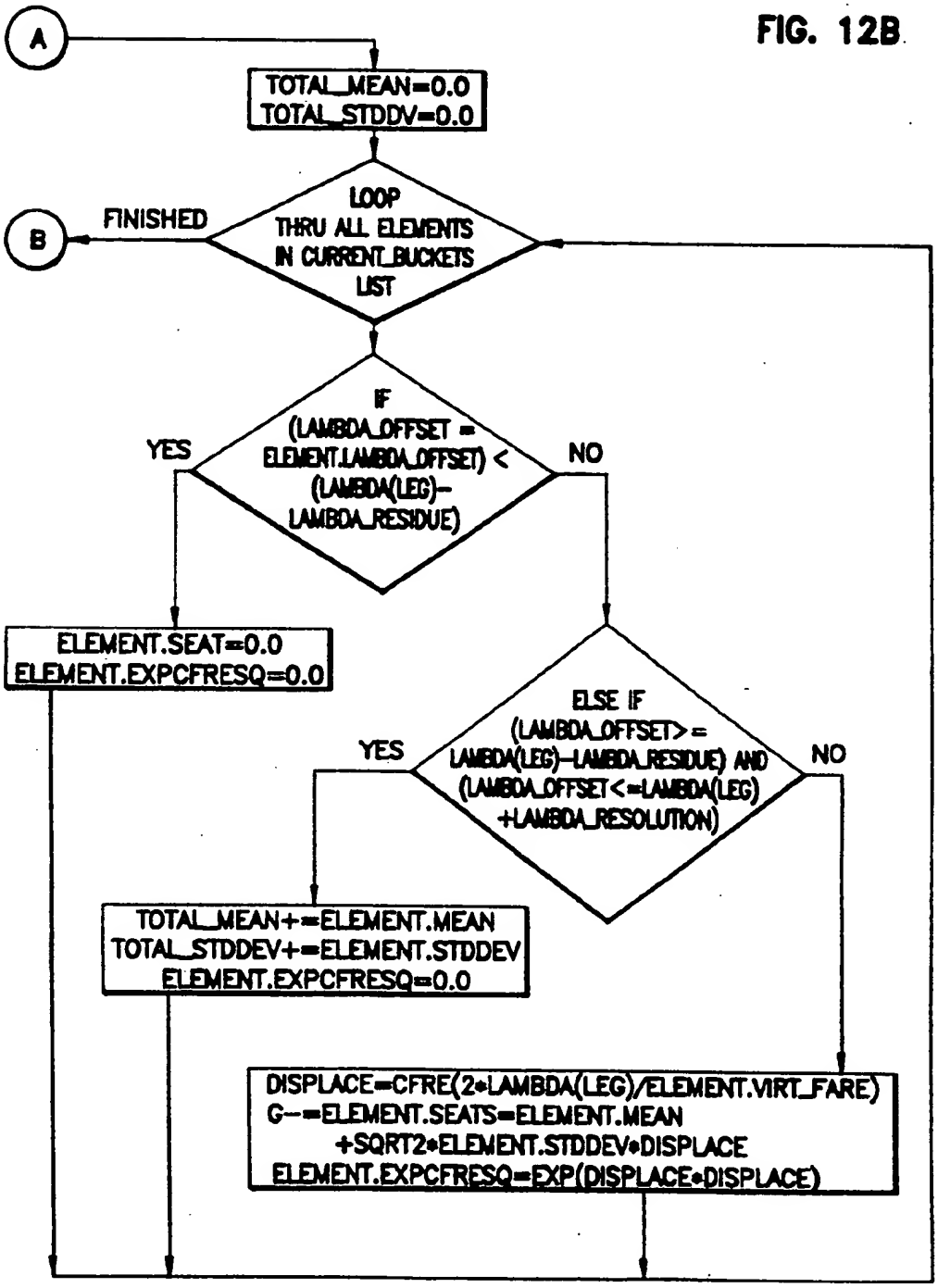

START
G=CAPACITY(LEG)
IF LAMBDA(LEG) < EPSILON
YES
NO
A
TOTAL_MEAN=0.0
TOTAL_STDDV=0.0
LOOP THRU ALL ELEMENTS IN CURRENT_BUCKETS LIST
FINISHED
IF ELEMENT.VIRT_FARE >=EPSILON
YES
NO
TOTAL_MEAN +=ELEMENT.MEAN
TOTAL_STDDEV +=ELEMENT.STDDEV*ELEMENT.STDDEV
TOTAL_STDDEV=SQRT(TOTAL_STDDEV)
G -=(TOTAL_MEAN+3*TOTAL_STDDEV)
IF (G>0.0) G=0.0
B
IF G>0.0
YES
NO
DISPLACE=(G-TOTAL_MEAN)/TOTAL_STDDEV
IF (DISPLACE >-2.5) DISPLACE=-2.5
LOOP THRU FIXED BUCKETS LIST
FINISHED
YES
G -=ELEMENT.SEATS*ELEMENT.MEAN + ELEMENT.STDDEV*DISPLACE
LOOP THRU FIXED BUCKETS LIST
FINISHED
YES
ELEMENT.SEATS=0.0
END

FIG. 12B


A
TOTAL_MEAN=0.0
TOTAL_STDDV=0.0
LOOP
THRU ALL ELEMENTS
IN CURRENT_BUCKETS
LIST
FINISHED
B
IF
(LAMBDA_OFFSET =
ELEMENT.LAMBDA_OFFSET) <
(LAMBDA(LEG)-
LAMBDA_RESIDUE)
YES
NO
ELEMENT.SEAT=0.0
ELEMENT.EXPCFRESQ=0.0
ELSE IF
(LAMBDA_OFFSET>=
LAMBDA(LEG)-LAMBDA_RESIDUE) AND
(LAMBDA_OFFSET<=LAMBDA(LEG)
+LAMBDA_RESOLUTION)
YES
NO
TOTAL_MEAN+=ELEMENT.MEAN
TOTAL_STDDEV+=ELEMENT.STDDEV
ELEMENT.EXPCFRESQ=0.0
DISPLACE=CFRE(2*LAMBDA(LEG)/ELEMENT.VIRT_FARE)
G--=ELEMENT.SEATS=ELEMENT.MEAN
+SQRT2*ELEMENT.STDDEV*DISPLACE
ELEMENT.EXPCFRESQ=EXP(DISPLACE*DISPLACE)

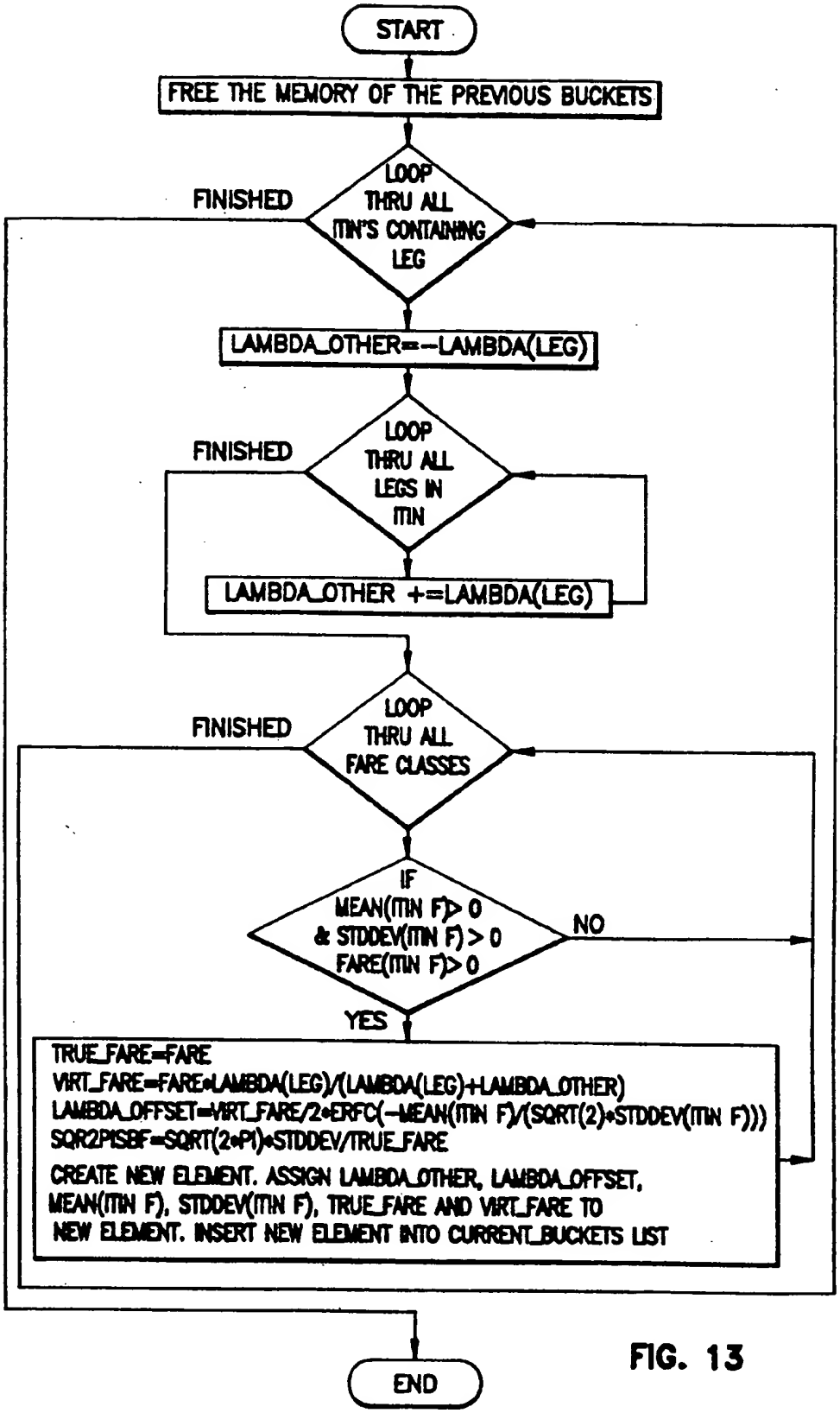

START
FREE THE MEMORY OF THE PREVIOUS BUCKETS
LOOP THRU ALL ITIN'S CONTAINING LEG
FINISHED
LAMBDA_OTHER=-LAMBDA(LEG)
LOOP THRU ALL LEGS IN ITIN
FINISHED
LAMBDA_OTHER +=LAMBDA(LEG)
LOOP THRU ALL FARE CLASSES
FINISHED
IF MEAN(ITIN F)> 0 & STDDEV(ITIN F) > 0 FARE(ITIN F)> 0
NO
YES
TRUE_FARE=FARE
VIRT_FARE=FARE*LAMBDA(LEG)/(LAMBDA(LEG)+LAMBDA_OTHER)
LAMBDA_OFFSET=VIRT_FARE/2*ERFC(-MEAN(ITIN F)/(SQRT(2)*STDDEV(ITIN F)))
SQR2PISBF=SQRT(2*PI)*STDDEV/TRUE_FARE
CREATE NEW ELEMENT. ASSIGN LAMBDA_OTHER, LAMBDA_OFFSET, MEAN(ITIN F), STDDEV(ITIN F), TRUE_FARE AND VIRT_FARE TO NEW ELEMENT. INSERT NEW ELEMENT INTO CURRENT_BUCKETS LIST
END

FIG. 13

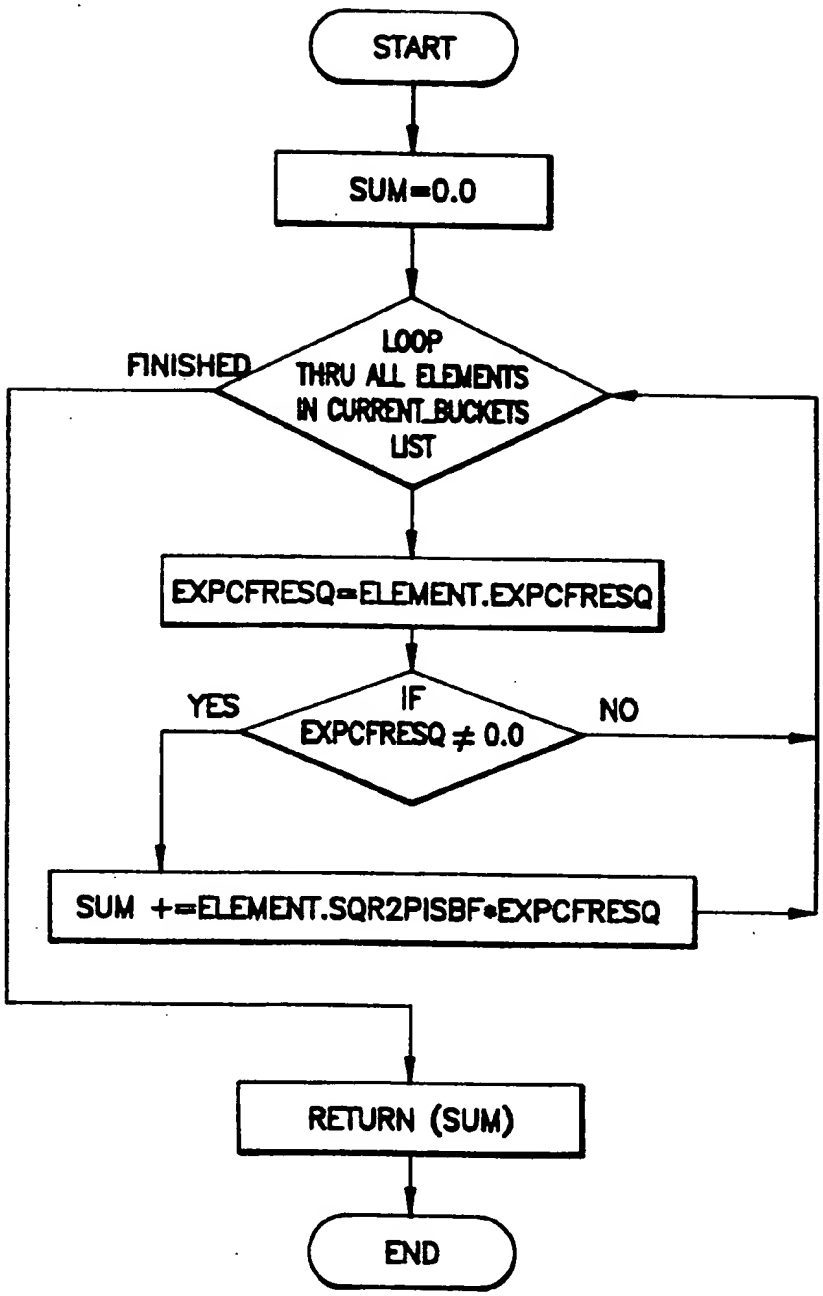

START
SUM=0.0
LOOP
THRU ALL ELEMENTS
IN CURRENT_BUCKETS
LIST
FINISHED
EXPCFRESQ=ELEMENT.EXPCFRESQ
IF
EXPCFRESQ ≠ 0.0
YES
NO
SUM +=ELEMENT.SQR2PISBF*EXPCFRESQ
RETURN (SUM)
END

FIG. 14

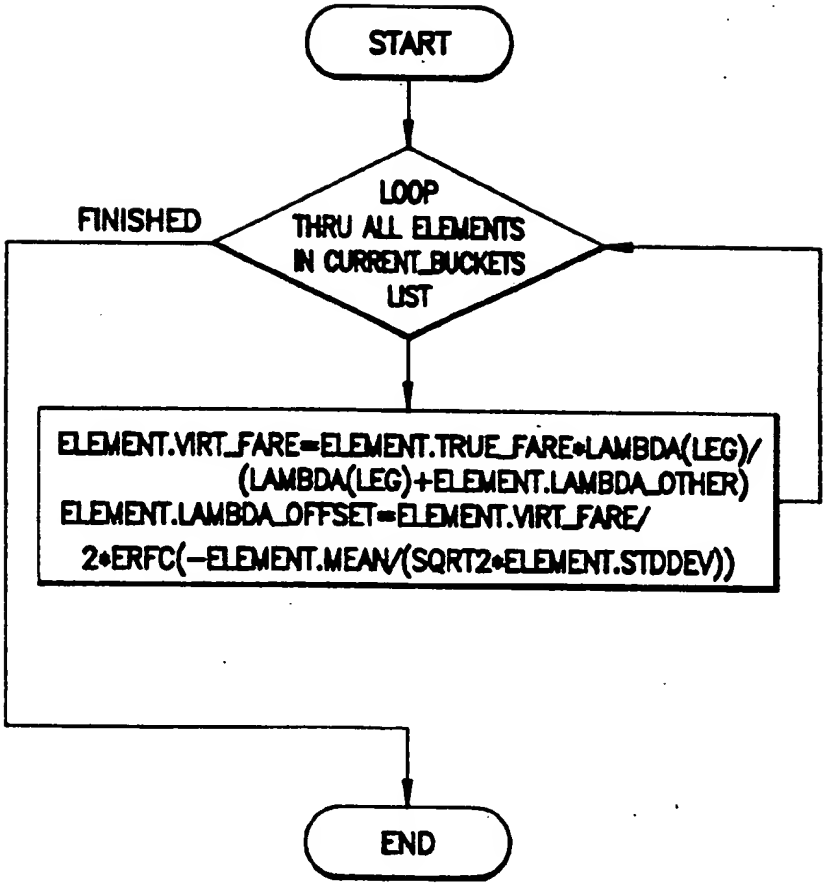

START
LOOP
THRU ALL ELEMENTS
IN CURRENT_BUCKETS
LIST
FINISHED
ELEMENT.VIRT_FARE=ELEMENT.TRUE_FARE*LAMBDA(LEG)/
(LAMBDA(LEG)+ELEMENT.LAMBDA_OTHER)
ELEMENT.LAMBDA_OFFSET=ELEMENT.VIRT_FARE/
2*ERFC(-ELEMENT.MEAN/(SQRT2*ELEMENT.STDDEV))
END

FIG. 15

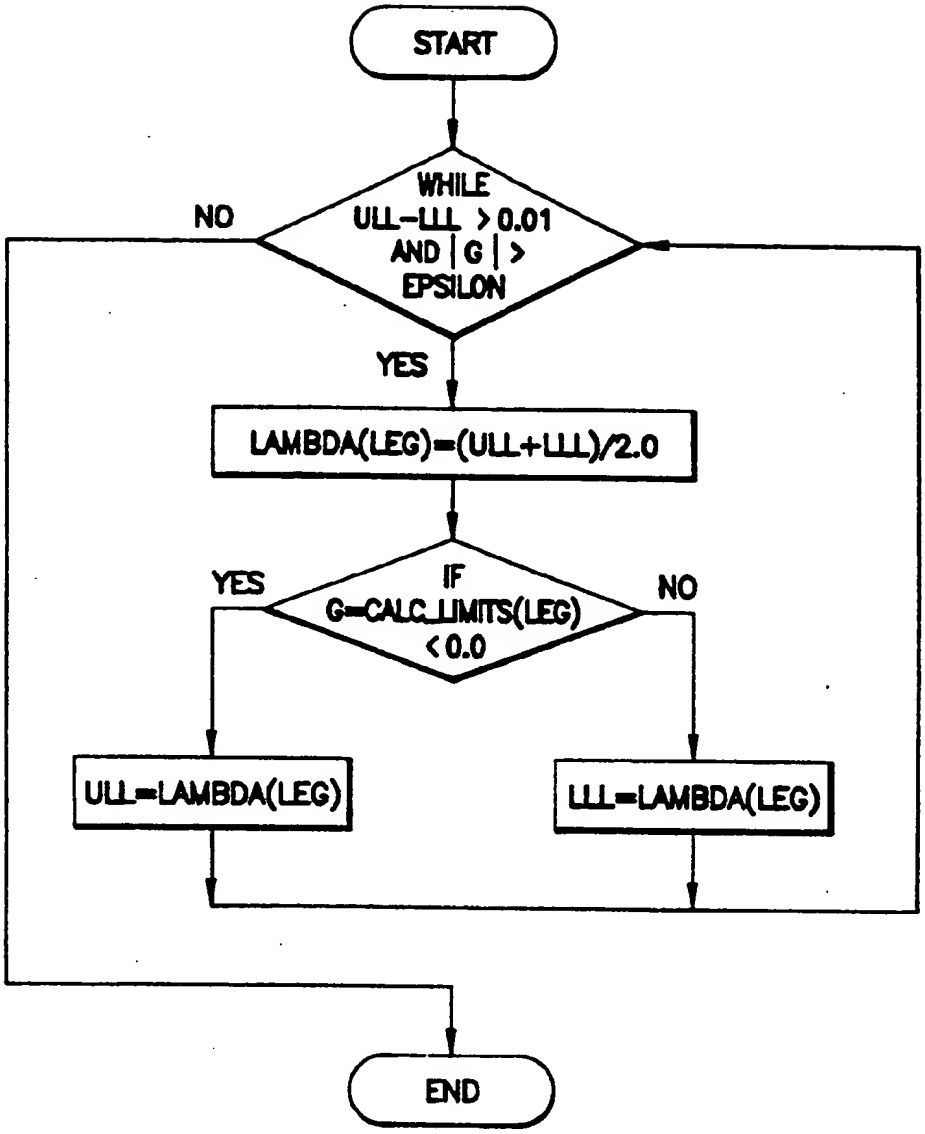

START
WHILE
ULL-LLL > 0.01
AND | G | >
EPSILON
NO
YES
LAMBDA(LEG)=(ULL+LLL)/2.0
IF
G=CALC_LIMITS(LEG)
< 0.0
YES
NO
ULL=LAMBDA(LEG)
LLL=LAMBDA(LEG)
END

FIG. 16

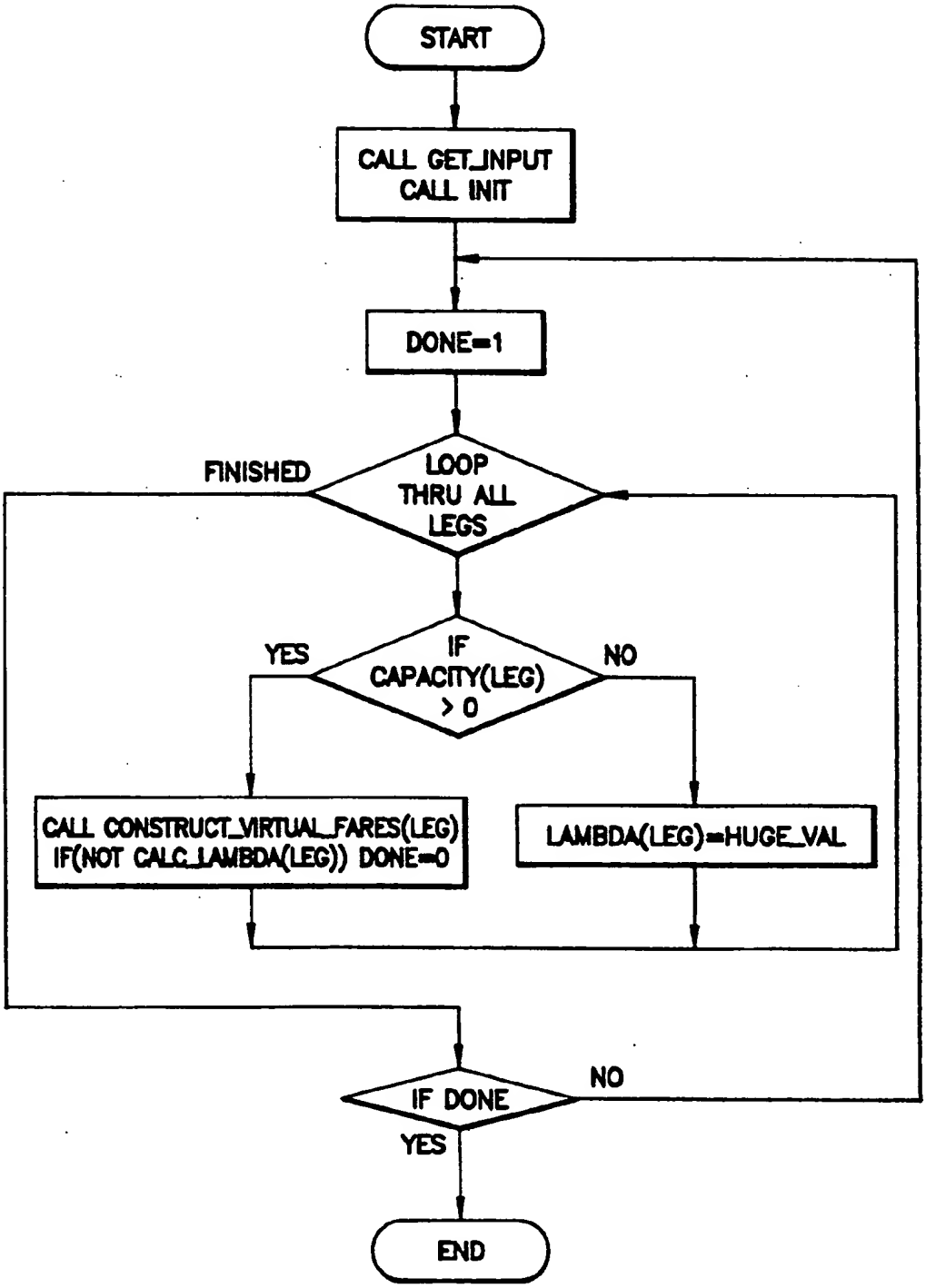

START
CALL GET_INPUT
CALL INIT
DONE=1
FINISHED
LOOP
THRU ALL
LEGS
YES
IF
CAPACITY(LEG)
> 0
NO
CALL CONSTRUCT_VIRTUAL_FARES(LEG)
IF(NOT CALC_LAMBDA(LEG)) DONE=0
LAMBDA(LEG)=HUGE_VAL
IF DONE
NO
YES
END

FIG. 17

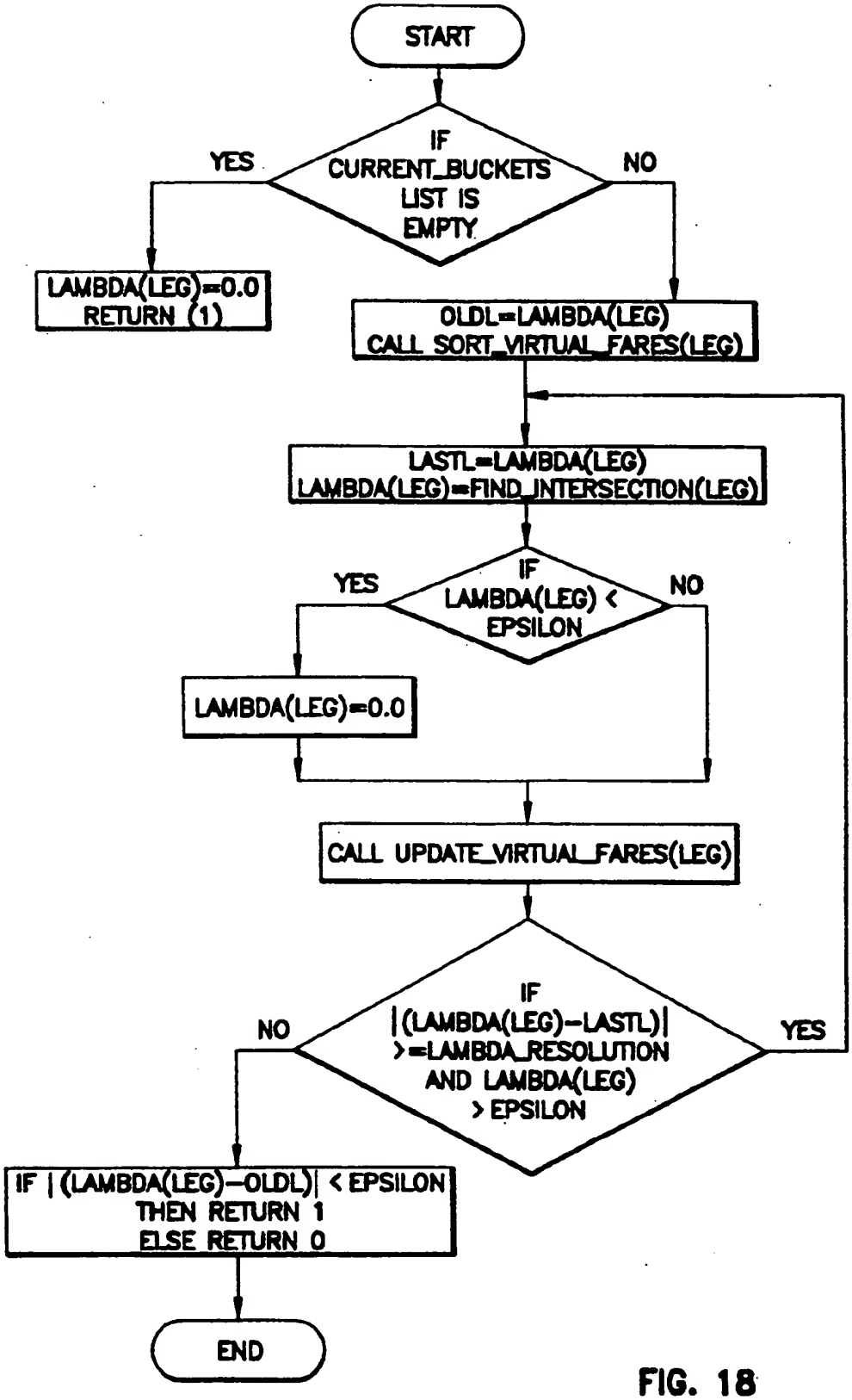

START
IF CURRENT_BUCKETS LIST IS EMPTY
YES
NO
LAMBDA(LEG)=0.0 RETURN (1)
OLDL=LAMBDA(LEG) CALL SORT_VIRTUAL_FARES(LEG)
LASTL=LAMBDA(LEG) LAMBDA(LEG)=FIND_INTERSECTION(LEG)
IF LAMBDA(LEG) < EPSILON
YES
NO
LAMBDA(LEG)=0.0
CALL UPDATE_VIRTUAL_FARES(LEG)
IF |(LAMBDA(LEG)-LASTL)| >=LAMBDA_RESOLUTION AND LAMBDA(LEG) > EPSILON
NO
YES
IF |(LAMBDA(LEG)-OLDL)| < EPSILON THEN RETURN 1 ELSE RETURN 0
END

FIG. 18

FIG. 19

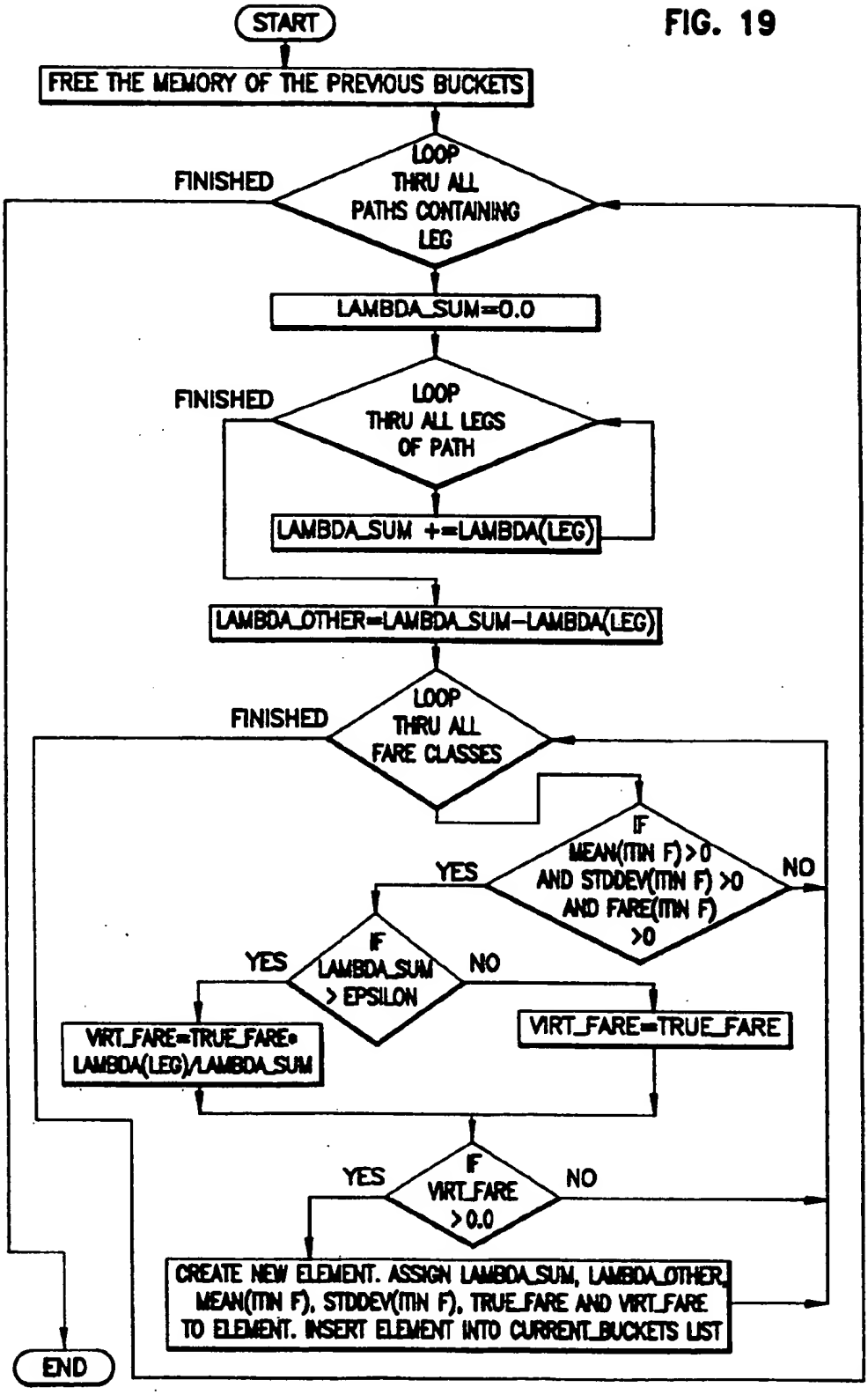

START
FREE THE MEMORY OF THE PREVIOUS BUCKETS
FINISHED
LOOP THRU ALL PATHS CONTAINING LEG
LAMBDA_SUM=0.0
FINISHED
LOOP THRU ALL LEGS OF PATH
LAMBDA_SUM +=LAMBDA(LEG)
LAMBDA_OTHER=LAMBDA_SUM-LAMBDA(LEG)
FINISHED
LOOP THRU ALL FARE CLASSES
IF MEAN(ITIN F)>0 AND STDDEV(ITIN F)>0 AND FARE(ITIN F) >0
YES
NO
IF LAMBDA_SUM > EPSILON
YES
NO
VIRT_FARE=TRUE_FARE* LAMBDA(LEG)/LAMBDA_SUM
VIRT_FARE=TRUE_FARE
IF VIRT_FARE >0.0
YES
NO
CREATE NEW ELEMENT. ASSIGN LAMBDA_SUM, LAMBDA_OTHER, MEAN(ITIN F), STDDEV(ITIN F), TRUE_FARE AND VIRT_FARE TO ELEMENT. INSERT ELEMENT INTO CURRENT_BUCKETS LIST
END

FIG. 20

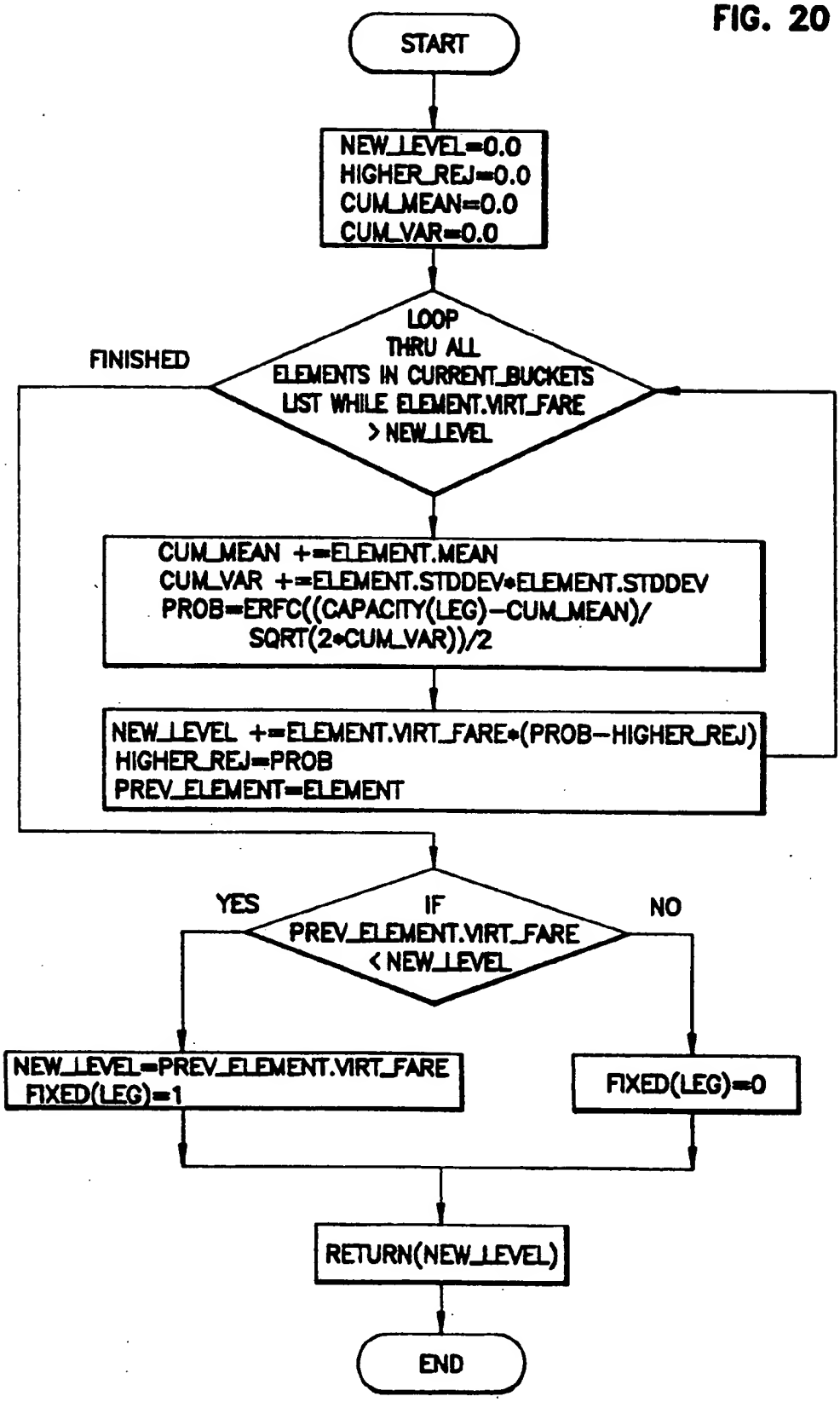
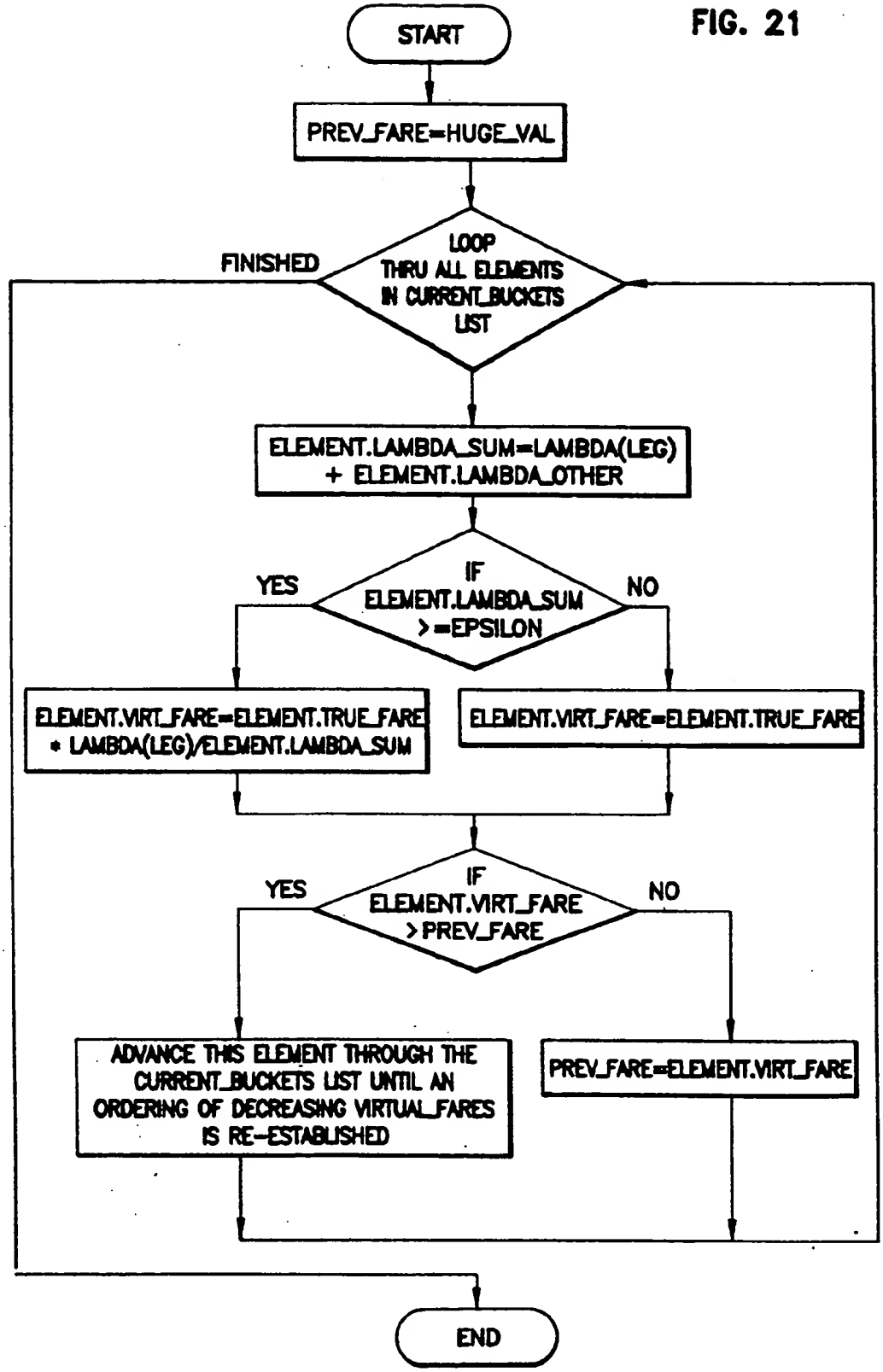

START
NEW_LEVEL=0.0
HIGHER_REJ=0.0
CUM_MEAN=0.0
CUM_VAR=0.0
LOOP
THRU ALL
ELEMENTS IN CURRENT_BUCKETS
LIST WHILE ELEMENT.VIRT_FARE
> NEW_LEVEL
FINISHED
CUM_MEAN +=ELEMENT.MEAN
CUM_VAR +=ELEMENT.STDDEV*ELEMENT.STDDEV
PROB=ERFC((CAPACITY(LEG)-CUM_MEAN)/
SQRT(2*CUM_VAR))/2
NEW_LEVEL +=ELEMENT.VIRT_FARE*(PROB-HIGHER_REJ)
HIGHER_REJ=PROB
PREV_ELEMENT=ELEMENT
IF
PREV_ELEMENT.VIRT_FARE
< NEW_LEVEL
YES
NO
NEW_LEVEL=PREV_ELEMENT.VIRT_FARE
FIXED(LEG)=1
FIXED(LEG)=0
RETURN(NEW_LEVEL)
END

FIG. 21


START
PREV_FARE=HUGE_VAL
LOOP THRU ALL ELEMENTS IN CURRENT_BUCKETS LIST
FINISHED
ELEMENT.LAMBDA_SUM=LAMBDA(LEG) + ELEMENT.LAMBDA_OTHER
IF ELEMENT.LAMBDA_SUM >=EPSILON
YES
NO
ELEMENT.VIRT_FARE=ELEMENT.TRUE_FARE * LAMBDA(LEG)/ELEMENT.LAMBDA_SUM
ELEMENT.VIRT_FARE=ELEMENT.TRUE_FARE
IF ELEMENT.VIRT_FARE >PREV_FARE
YES
NO
ADVANCE THIS ELEMENT THROUGH THE CURRENT_BUCKETS LIST UNTIL AN ORDERING OF DECREASING VIRTUAL_FARES IS RE-ESTABLISHED
PREV_FARE=ELEMENT.VIRT_FARE
END

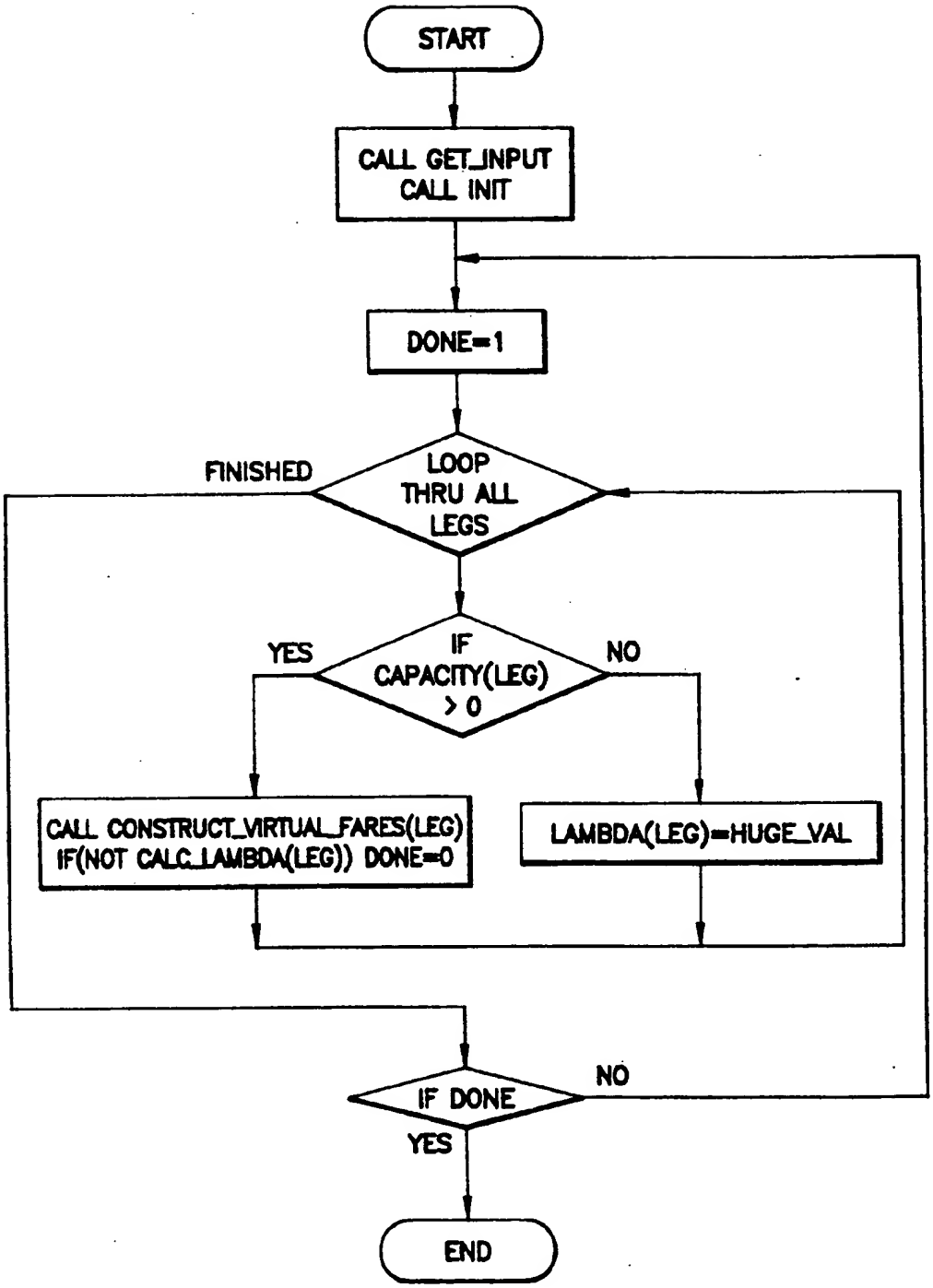

START
CALL GET_INPUT
CALL INIT
DONE=1
LOOP THRU ALL LEGS
FINISHED
IF CAPACITY(LEG) > 0
YES
NO
CALL CONSTRUCT_VIRTUAL_FARES(LEG)
IF(NOT CALC_LAMBDA(LEG)) DONE=0
LAMBDA(LEG)=HUGE_VAL
IF DONE
NO
YES
END

FIG. 22

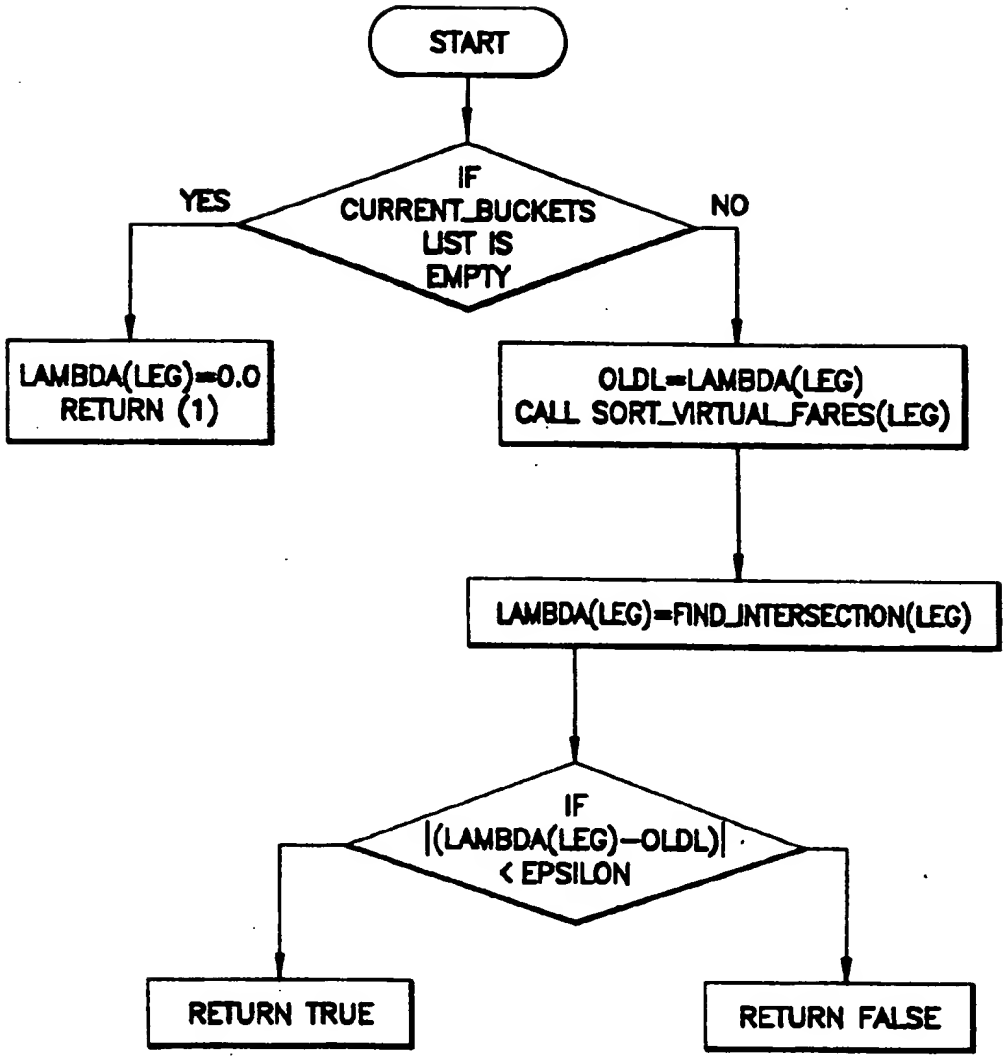

START
IF
CURRENT_BUCKETS
LIST IS
EMPTY
YES
NO
LAMBDA(LEG)=0.0
RETURN (1)
OLDL=LAMBDA(LEG)
CALL SORT_VIRTUAL_FARES(LEG)
LAMBDA(LEG)=FIND_INTERSECTION(LEG)
IF
|(LAMBDA(LEG)-OLDL)|
< EPSILON
RETURN TRUE
RETURN FALSE

FIG. 23

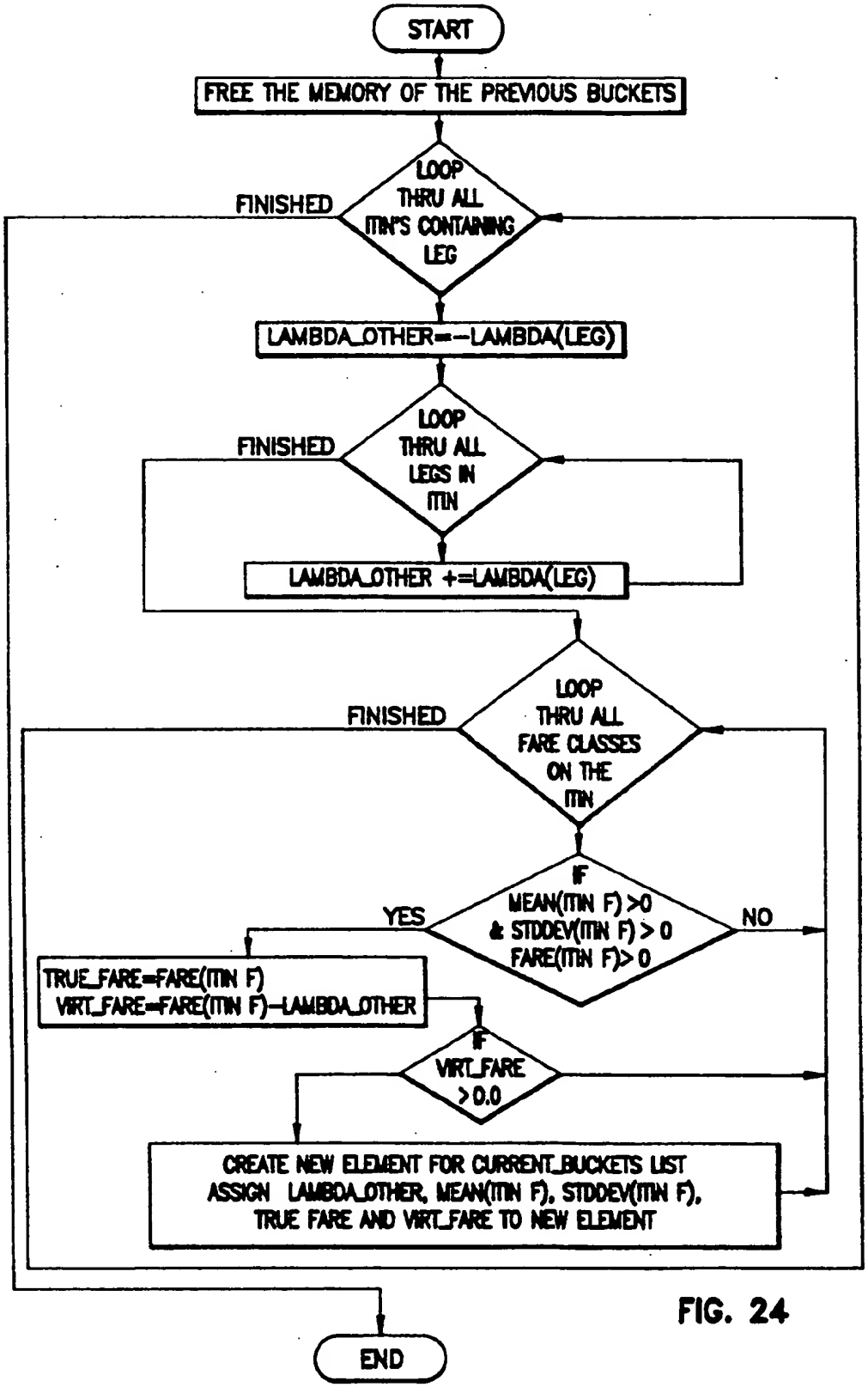
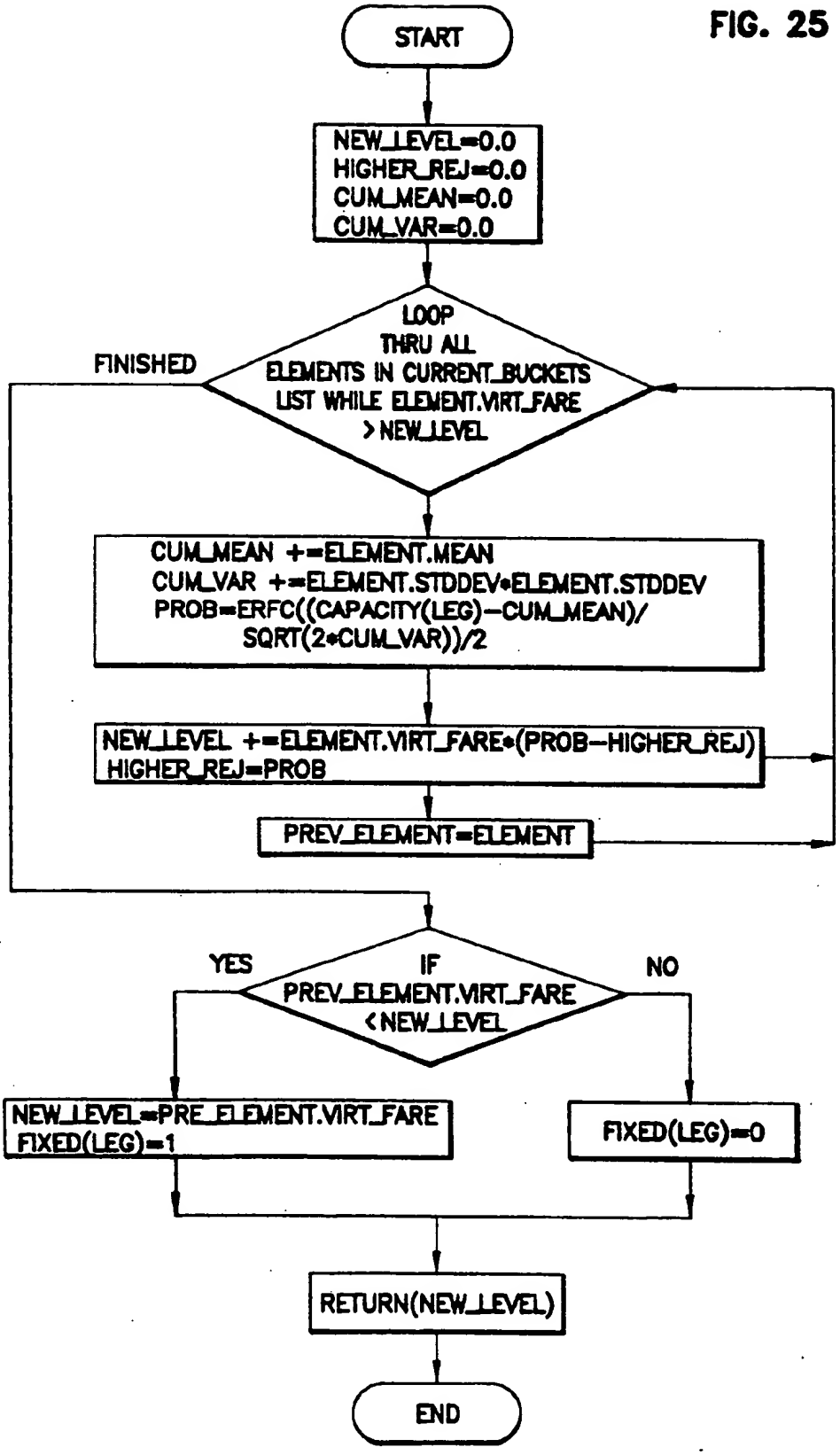

START
FREE THE MEMORY OF THE PREVIOUS BUCKETS
FINISHED
LOOP THRU ALL ITIN'S CONTAINING LEG
LAMBDA_OTHER=-LAMBDA(LEG)
FINISHED
LOOP THRU ALL LEGS IN ITIN
LAMBDA_OTHER +=LAMBDA(LEG)
FINISHED
LOOP THRU ALL FARE CLASSES ON THE ITIN
IF MEAN(ITIN F) >0 & STDDEV(ITIN F) > 0 FARE(ITIN F)> 0
YES
NO
TRUE_FARE=FARE(ITIN F)
VIRT_FARE=FARE(ITIN F)-LAMBDA_OTHER
IF VIRT_FARE > 0.0
CREATE NEW ELEMENT FOR CURRENT_BUCKETS LIST
ASSIGN LAMBDA_OTHER, MEAN(ITIN F), STDDEV(ITIN F),
TRUE FARE AND VIRT_FARE TO NEW ELEMENT
END

FIG. 24

FIG. 25



12/30/2002, EAST Version: 1.03.0002

VIRTUAL FARE METHODS FOR A COMPUTERIZED AIRLINE SEAT INVENTORY CONTROL SYSTEM

BACKGROUND OF THE INVENTION

1. Cross Reference To Related Applications

This application is a continuation-in-part of patent application Ser. No. 07/630,261 filed Dec. 19, 1990 by Scot W. Hornick et al. entitled "AIRLINE SEAT INVENTORY CONTROL METHOD AND APPARATUS FOR COMPUTERIZED AIRLINE RESERVATION SYSTEMS", which application is incorporated herein by reference.

2. Field Of The Invention

This invention relates generally to an airline reservation system. In particular, the present invention provides an airline seat inventory control system for computerized airline reservation systems.

3. Description Of Related Art

Strategic and operational planning for commercial airlines are highly complicated problems, especially since the industry has been deregulated. In order to cope with this complexity, computer-based decision support systems have been adopted to facilitate the planning of schedules, routes, aircraft and crew rotations and yield management. Yield (or revenue) management is one of the most important aspects of the operational plan for a commercial airline. Yield management can be separated into two distinct parts: pricing and seat inventory control. Pricing involves the establishment of fare classes and tariffs within those classes for each specific origin-destination market. Seat inventory control is the periodic adjustment of nested booking limits for the various fare classes so as to optimize the passenger mix and thereby maximize the generated revenue. In particular, the objective is to fly the aircraft as full as possible without allowing the earlier-booking, discount-fare passengers to displace the later-booking, full-fare passengers.

Recently, considerable research has been devoted to developing automated seat inventory control methods (For a survey, see the following publications, all of which are incorporated herein by reference: P. P. Belobaba, "Airline yield management, an overview of seat inventory control," Transportation Science, 21 (1987), no. 2, pp. 63–72; For a comparative evaluation see E. L. Williamson, "Comparison of the optimization techniques for origin-destination seat inventory control," Technical Report FTL-R88-2, Flight Transportation Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., May 1988). However, the proposed methods all have serious limitations.

Some methods are leg-based and therefore do not produce booking limits that are optimal in a system-wide sense. For example, the "locally greedy" approach used by these methods may not recognize the additional revenue generated by long-haul (multi-leg-itinerary) passengers versus short-haul (single-leg-itinerary) passengers, or, on the other hand, they may have an uneconomical bias to long-haul passengers (see, e.g., the following publications, all of which are incorporated herein by reference: K. Littlewood, "Forecasting and control of passenger bookings," Proceedings of the 12th AGIFORS Symposium, 1972, pp. 95–117; A. V. Bhatia and S. C. Parekh, "Optimal allocation of seats by fare," Presentation to the AGIFORS Reservation Study Group, 1973; H. Richter, "The differential revenue method to determine optimal seat allotments by fare type," Proceedings of the 22nd AGIFORS Symposium, 1982, pp. 339–362; P. P. Belobaba, "Air travel demand and airline seat inventory management," Technical Report FTL-R87-8, Flight Transportation Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., May 1987; P. P. Belobaba, "Application of a probabilistic decision model to airline seat inventory control:, Operations Research, 37 (1989), No. 2, pp. 183–197).

Other methods are network-based, but assume a deterministic demand model, i.e., they assume that demand for air travel in a particular market is known precisely without any uncertainty (see, e.g., the following publication, which is incorporated herein by reference: F. Glover, R. Glover, J. Lorenzo, and C. McMillan, "The passenger-mix problem in the scheduled airlines," Interfaces, 12 (1982), pp. 73–79). Such methods do not reserve enough seats to capture higher-than-average demand for the more expensive fare classes. Further, these methods use linear programming formulations with large numbers of variables (and concomitantly time-consuming solutions) to determine the booking limits for each fare class. Efforts to simultaneously achieve network-wide optimality and account for the probabilistic nature of demand have resulted in 0–1 integer programming formulations with an even larger number of variables (see, e.g., the following publication, which is incorporated herein by reference: R. D. Wollmer, "An airline reservation model for opening and closing fare classes," Unpublished Internal Report, McDonnell-Douglas Corporation, Long Beach, Calif., 1985). The large number of variables and the complexity of the solution methods make these approaches unsuitable for real-world problems.

SUMMARY OF THE INVENTION

To overcome the limitations in the prior art discussed above, and to overcome other limitations readily recognizable to those skilled in the art, the present invention discloses an airline reservation system wherein reservations are controlled using, in part, a seat inventory control system. The present invention provides an airline seat reservation system that produces optimal network-wide seat inventory controls while taking into account the probabilistic nature of demand. The present invention, based on a concept termed Network-Based Expected Marginal Seat Revenue (EMSR), does not require the large number of variables required by the other network-based approaches, and it incorporates a probabilistic demand model without resorting to computationally intractable integer programming.

In the present invention, a computer-based seat inventory control system uses iterative leg-based methods to control bookings in a flight network comprised of a plurality of itinerary/fare class combinations using a plurality of flight legs. When considering a particular flight leg, the total fare paid by a passenger using the leg is adjusted by taking into account an estimate of the displacement cost of the travel on the other legs of the itinerary to create a virtual fare. Expected marginal seat revenues (or more precisely, their current approximations) provide the displacement costs on the legs when computing virtual fares. Using these virtual fares, a leg-based optimization method is applied to the individual legs one-by-one to compute new approximations of the expected marginal seat revenues. This method is

iterated until the expected marginal seat revenues converge to their network-optimal values.

BRIEF DESCRIPTION OF THE DRAWINGS

In the drawings, where like numerals refer to like elements throughout the several views:

FIG. 1 is a block diagram describing the components of an integrated operation and strategic planning system using the inventory control method and apparatus of the present invention;

FIG. 2 is a flow chart describing the logic of an iterated leg-based method;

FIG. 3 is an illustration of two cases for the solution of Equation (8) infra;

FIG. 4 is a flow chart describing the logic of an iterated leg-based method for EMSR-prorated virtual fares;

FIG. 5 is a flow chart describing the logic of the GET INPUT routine;

FIG. 6 is combined are a flow chart describing the logic of the INIT routine;

FIG. 7 is a flow chart describing the logic of the CONSTRUCT MILE PRORATED FARES routine in the unnested EMSR-prorated virtual fare method;

FIG. 8 is a flow chart describing the logic of the CONSTRUCT MILE PRORATED FARES routine in the nested EMSR-prorated virtual fare method;

FIG. 9 is a flow chart describing the logic of the CONSTRUCT MILE PRORATED FARES routine in the nested EMSR-differential virtual fare method;

FIG. 10 is a flow chart describing the MAIN routine in the unnested EMSR-prorated virtual fare method;

FIGS. 11A and 11B combined are a flow chart describing the CALC LAMBDA routine in the unnested EMSR-prorated virtual fare method;

FIGS. 12A and 12B combined are a flow chart describing the CALC LIMITS routine in the unnested EMSR-prorated virtual fare method;

FIG. 13 is a flow chart describing the CONSTRUCT VIRTUAL FARES routine in the unnested EMSR-prorated virtual fare method;

FIG. 14 is a flow chart describing the DELFUNC routine;

FIG. 15 is a flow chart describing the UPDATE VIRTUAL FARES routine in the unnested EMSR-prorated virtual fare method;

FIG. 16 is a flow chart describing the BINARY routine;

FIG. 17 is a flow chart describing the MAIN routine in the nested EMSR-prorated virtual fare method;

FIG. 18 is a flow chart describing the CALC LAMBDA routine in the nested EMSR-prorated virtual fare method;

FIG. 19 is a flow chart describing the CONSTRUCT VIRTUAL FARES routine in the nested EMSR-prorated virtual fare method;

FIG. 20 is a flow chart describing the FIND INTERSECTION routine in the nested EMSR-prorated virtual fare method;

FIG. 21 is a flow chart describing the operation of the UPDATE VIRTUAL FARES routine in the nested EMSR-prorated virtual fare method;

FIG. 22 is a flow chart describing the MAIN routine in the nested EMSR-differential virtual fare method;

FIG. 23 is a flow chart describing the CALC LAMBDA routine in the nested EMSR-differential virtual fare method;

FIG. 24 is a flow chart describing the CONSTRUCT VIRTUAL FARES in the nested EMSR-differential virtual fare method; and

FIG. 25 is a flow chart describing the FIND INTERSECTION routine in the nested EMSR-differential virtual fare method.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

In the following Detailed Description of the Preferred Embodiment, reference is made to the accompanying Drawings which form a part hereof, and in which is shown by way of illustration three embodiments in which the invention may be practiced. It is to be understood that other embodiments may be utilized, and changes may be made to both the that process and the structure without departing from the scope of the present invention.

The present invention is an airline seat reservation system wherein seat reservations are controlled using, in part, a seat inventory control system. The present invention provides optimal seat reservation control using network-wide booking controls while taking into account the probabilistic nature of demand. The seat reservation control, based on a concept termed Network-Based Expected Marginal Seat Revenue (EMSR), does not require the large number of variables required by the other network-based approaches, and it incorporates a probabilistic demand model without resorting to computationally intractable integer programming as compared in Table 1.

This Detailed Description is organized into eight sections. In the first section, entitled "An Airline Seat Reservation System", the basic components of an airline seat reservation system are described, illustrating how the seat inventory control system can be integrated into such a system. The seat inventory control system generates the information necessary to set booking limits for the airline seat reservation system.

In the second section, entitled "A Mathematical Formulation", the mathematical formulation of the unnested network-based seat inventory control problem as a constrained nonlinear optimization problem is described. The optimum is characterized by a relatively small system of nonlinear equations.

In the third section, entitled "Solving The System With Virtual Fares And A Leg-Based Method," it is described how the system of equations characterizing the unnested network optimum can be solved by iterating a leg-based method and using EMSR-dependent virtual fares.

In the fourth section, entitled "Incorporating Nesting In The Optimization Model," the uses of nested availability in the optimization model are described.

In the fifth through eighth sections, the implementation of three embodiments of the airline seat inventory control is described.

I. An Airline Seat Reservation System

FIG. 1 describes the elements of an airline seat reservation system 1 according to the present invention. The reservation system 1 is a software system that is executed by a computer 2. The reservation system 1 communicates with a plurality of remote terminals 3 and printers 4. The reservation system 1 accepts or rejects seat reservation requests and records those reservations 10 in a database 9. In the present invention, a seat inventory control software system 5 uses a flight network

database 6 to generate seat booking limits 7 and/or expected marginal seat revenues (EMSRs) 8 for input to the reservation system 1. The reservation system 1 relies on this data 7 and 8 to determine whether to accept or reject seat reservation requests.

The database 6 describes a flight network comprising a plurality of flight legs a, and itinerary p and fare class i combinations. Each flight leg a has a residual seating capacity C_a, and each itinerary p and fare class i combination has a revenue yield f_p^i for a seat reserved therein.

Using an unnested EMSR-prorated virtual fare method of allocating seats in a computerized airline reservation system, the system calculates an initial expected marginal seat revenue (EMSR) λ_a 8 for all flight legs a. An unnested EMSR-prorated virtual fare $v_{p,a}^i$ is computed for every itinerary p and fare class i combination that contains a particular flight leg a having a nonzero residual seating capacity C_a so that:

$$v_{p,a}^i \triangleq \frac{f_p^i \lambda_a}{\sum_{b \in p} \lambda_b}$$

A new EMSR λ_a 8 for the particular flight leg a is calculated based on the virtual fares $v_{p,a}^i$ by applying Newton's method to a seating capacity constraint for the particular flight leg a:

$$\sum_{\substack{p \\ a \in p \\ v_{p,a}^i > \lambda_a}} \sum_i Q_p^i(\lambda_a / v_{p,a}^i) = C_a$$

thereby ensuring that a total number of seats assigned to the itinerary p and fare class i combinations are equal to the residual seating capacity of the particular flight leg a, wherein the virtual fares $v_{p,a}^i$ are updated at each step of the Newton's method since each step changes the EMSR λ_a for the particular flight leg a. The above steps are repeated for the particular flight leg a until the EMSR λ_a 8 converges. The above steps are repeated for all flight legs a until the changes in EMSR's λ_a's 8 therefor are insignificant.

Using a nested EMSR-prorated virtual fare method of allocating seats in a computerized airline seat inventory control system, the system calculates an initial expected marginal seat revenue (EMSR) λ_a 8 for all the flight legs a. A nested EMSR-prorated virtual fare $v_{p,a}^i$ is computed for each itinerary p and fare class i combination that contains a particular flight leg a having a nonzero residual seating capacity C_a so that:

$$v_{p,a}^i \triangleq \frac{f_p^i \lambda_a}{\sum_{b \in p} \lambda_b}$$

The itinerary p and fare class i combinations are sorted into a list ordered by descending values of virtual fares $v_{p,a}^i$. The sorted list of virtual fares $v_{p,a}^i$ is processed one-by-one to find an intersection point defining a new EMSR λ_a 8 for the particular flight leg a between functions:

$$\lambda_a = x$$

-continued

$$\lambda_a = \sum_{f_d^j \geq x} f_d^j (\pi_j - \pi_{j-1})$$

wherein x is an independent variable, f_d^j is a jth largest virtual fare on leg a, and π_j is a probability that more than C_a passengers are willing to pay f_d^j or more to travel on leg a. The above steps are repeated for the particular flight leg a until the EMSR λ_a 8 converges. The above steps are repeated for all flight legs a until the changes in EMSR's λ_a's therefor are insignificant.

Using a nested EMSR-differential virtual fare method of allocating seats in a computerized airline seat inventory control system, the system calculates an initial expected marginal seat revenue (EMSR) λ_a 8 for all the flight legs a. A nested EMSR-differential virtual fare $v_{p,a}^i$ is computed for each itinerary p and fare class i combination that contains a particular flight leg a having a nonzero residual seating capacity C_a so that:

$$v_{p,a}^i \triangleq f_p^i - \sum_{\substack{b \in p \\ b \neq a}} \lambda_b$$

The itinerary p and fare class i combinations are sorted into a list ordered by descending virtual fares $v_{p,a}^i$. The sorted list of virtual fares $v_{p,a}^i$ is processed one-by-one to find an intersection point defining a new EMSR λ_a 8 for the particular flight leg a between functions:

$$\lambda_a = x$$

$$\lambda_a = \sum_{f_d^j \geq x} f_d^j (\pi_j - \pi_{j-1})$$

wherein x is an independent variable, f_d^j is a jth largest virtual fare on leg a, and π_j is a probability that more than C_a passengers are willing to pay the virtual fare f_d^j or more to travel on leg a. The above steps are repeated for all flight legs a until changes in EMSR's λ_a's 8 therefor are insignificant.

Operators at the reservation terminals 3 enter a seat reservation request for a particular itinerary/fare class. The computer 2 receives the seat reservation request from the reservation terminals 3 and the airline reservation system 1 accepts or rejects the seat reservation request.

Two different paradigms may be used for determining availability, i.e., allocation-based availability and value-based availability. In allocation-based availability, bookings are compared to a booking limit or authorization level, so that the seat reservation request is accepted when the total number of seats 7 assigned to the itinerary p and fare class i combinations is not exceeded. In value-based availability, the generated revenue is compared to a sum of expected marginal seat revenues, so that the seat reservation request is accepted when the seat reservation request would yield revenue greater than or equal to a sum of the EMSR's λ_a's 8 for all flight legs a in the itinerary p.

A electronic status message indicating acceptance or rejection of the seat reservation request is transmitted by the airline seat reservation system 1 to the reservation terminal 3. If accepted, the seat reservation request is recorded in the database 9 and a ticket may be printed on the printer 4 attached to the terminal 3.

II. A Mathematical Formulation

1. Deriving the Network Equations

The network-based seat inventory control problem can be formulated on a directed graph D=(V,A), where the set of vertices V represents the set of cities and the set of arcs A represents the set of flight legs. If the seat inventory control system is controlling bookings for a particular day, then A represents the set of all flight legs occurring on that day, and it may include more than one flight leg between some city pairs. A path or itinerary p has a sequence of connecting flight legs in the graph D with an origin city s(p) and a destination city t(p). If leg a is included in itinerary p, then $a \epsilon p$. In this context, only itineraries satisfying minimum connection times constraints and having non-negligible traffic demand will be considered. The set of all such itineraries is denoted by P, and for each $p \epsilon P$, f^i_p is established, which is the nominal revenue yield or tariff from fare class i for itinerary p travel from city s(p) to city t(p). Booking limits S^i_p must be set for fare class i on itinerary p so as to maximize total system revenue, subject to the constraint that the total number of seats authorized for sale on each flight leg a is exactly equal to the capacity of that leg C_a, i.e., the number of seats on the aircraft flying the leg (overbooking is considered in the parent patent application incorporated herein by reference). As a matter of notation, the superscript i is dropped whenever a total over all fare classes is taken, and hence:

$$S_p \triangleq \sum_{i=1}^{\Phi} S_p^i$$

where Φ is the number of fare classes or tariff code buckets.

Since demand is stochastic, the total system revenue is actually a random variable, and so the objective of the present invention is to maximize its expected value R, which is just the sum of the expected revenue for all itinerary/fare class combinations. Writing this as an equation gives:

$$R = \sum_{p \epsilon P} \sum_{i=1}^{\Phi} R_p^i(S_p^i) \tag{1}$$

The capacity constraint for leg a can be written as:

$$\sum_{\substack{p \epsilon P \\ a \epsilon p}} S_p = C_a, \text{ for all } a \epsilon A \tag{2}$$

In the parent patent application Ser. No. 07/630,261 filed Dec. 19, 1990 by Scot W. Hornick et al. entitled "AIRLINE SEAT INVENTORY CONTROL METHOD AND APPARATUS FOR COMPUTERIZED AIRLINE RESERVATION SYSTEMS", incorporated herein by reference, it was shown that the solution to this constrained optimization problem is characterized by the system of equations:

$$\sum_{\substack{p \epsilon P \\ a \epsilon p}} \sum_{i=1}^{\Phi} Q_p^i\left(\sum_{b \epsilon p} \lambda_b / f_p^i\right) = C_a, \text{ for all } a \epsilon A \tag{3}$$

wherein λ_b is the expected marginal seat revenue (EMSR) for leg b and $Q^i_p(.)$ is the inverse of the cumulative probability density function of demand for fare class i travel on itinerary p.

III. Solving The System With Virtual Fares And A Leg-Based Method

The notion of using a leg-based method to control bookings and yet somehow accounting for the differences between long-haul and short-haul demand has given rise to virtual nesting methods (see, e.g., the following publication, which is incorporated herein by reference: P. P. Belobaba, "Air travel demand and airline seat inventory management," Technical Report FTL-R87-8, Flight Transportation Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., May 1987). When considering a particular flight leg, these methods discount the total fare paid by a long-haul passenger using this leg by an estimate of the displacement cost of his travel on the other legs of his itinerary. Using these discounted or virtual fares, virtual nesting methods aggregate itinerary/fare class combinations with similar virtual fares into virtual fare buckets, which are then controlled by a conventional leg-based method. Large virtual nesting tables must be maintained in order to map the numerous itinerary/fare class combinations into different virtual fare buckets on each leg.

An important aspect of the present invention is the observation that it is unnecessary to use estimates of the displacement cost on the legs when computing virtual fares; the EMSR's themselves are the displacement costs. Hence, the virtual fare for travel on leg a along itinerary p at fare class i is:

$$v^i_{p,a} \triangleq f^i_p - \sum_{\substack{b \epsilon p \\ b \neq a}} \lambda_b \tag{4}$$

This is termed an EMSR-differential virtual fare since it is computed by taking the difference between the total fare 15 and the EMSR's on the other legs. A leg-based method that uses EMSR-differential virtual fares must be iterative since, as it recomputes the EMSR's, it also changes the virtual fares upon which they were based.

As was indicated, an iterated leg-based method using virtual fares can be used to solve Eq. (3). However, the EMSR-differential virtual fares of Eq. (4) will not ensure this equivalence. In the unnested network-based method of the parent patent application Ser. No. 07/630,261 filed Dec. 19, 1990 by Scot W. Hornick et al. entitled "AIRLINE SEAT INVENTORY CONTROL METHOD AND APPARATUS FOR COMPUTERIZED AIRLINE RESERVATION SYSTEMS", incorporated herein by reference, the number of seats allocated to itinerary p fare class i is:

$$S_p^i = Q_p^i\left(\sum_{b \epsilon p} \lambda_b / f_p^i\right) \tag{5}$$

Let $S^i_{p,a}$ be the number of seats assigned to itinerary p fare class i travel on leg a by a leg-based method using virtual fares $v^i_{p,a}$. The formula for $S^i_{p,a}$ (using a distinct allocation method) is:

$$S^i_{p,a} = Q^i_p(\lambda_a / V^i_{p,a}) \tag{6}$$

Equating Eqs. (5) and (6), it can be shown that equivalence is achieved when:

$$\lambda_a / V^i_{p,a} = \sum_{b \in p} \lambda_b / f_p^i \quad (7)$$

or

$$V^i_{p,a} = \frac{f_p^i \lambda_a}{\sum_{b \in p} \lambda_b}$$

This is termed an EMSR-prorated virtual fare. FIG. 2 is a flow chart illustrating that when an iterated leg-based method using EMSR-prorated virtual fares converges, the seat allocations and EMSRs coincide with the optimal network-based solution, i.e., the solution of the system in Eq. (3).

IV. Incorporating Nesting In The Optimization Model

In a previous section, a nested availability rule based on the sum of the EMSRs along a passenger itinerary was discussed. On the other hand, all of the optimization models discussed so far are based on, or internally maintain, separate and distinct seat inventories for each itinerary/fare class combination. While these two ideas are apparently contradictory, an unnested optimization model can be used quite effectively with a nested availability rule. Nevertheless, some additional benefit can be derived from incorporating nesting into the optimization model as well.

This can be done by developing a nested leg-based optimization method and applying it to the EMSR-prorated virtual fare technique to obtain network-optimal results. Consider a leg a with virtual fares:

$$F^1_a \geq f^2_a \geq \ldots \geq f^m_a$$

Let E_i denote the event that the demand for virtual fares f^i_a and above exceeds the capacity C_a, i.e., E_i occurs if more than C_a passengers are willing to pay f^i_a or more to travel on leg a. $P(E_i) = \pi_i$ is the probability of an occurrence of event E_i. Given that a seat will not be sold to a passenger who is unwilling to pay at least λ_a (viewing λ_a as a control parameter for the moment), the expected incremental value of a seat on the leg is:

$$\sum_{f_a^i \geq \lambda_a} f_a^i P(E_i \cap \bar{E}_{i-1})$$

where $E_0 = \emptyset$ and "—" denotes complementation. However, this is just the EMSR. Thus, the correct value of λ_a must satisfy:

$$\lambda_a = \sum_{f_a^i \geq \lambda_a} f_a^i P(E_i \cap \bar{E}_{i-1})$$

Noting that $E_{i-1} \subseteq E_i$ by definition, this can be rewritten as:

$$\lambda_a = \sum_{f_a^i \geq \lambda_a} f_a^i (\pi_i - \pi_{i-1})$$

Solving this equation can be thought of as finding the intersection of the graphs of two functions of an independent variable x:

$$\lambda_a = x \quad (8)$$

$$\lambda_a = \sum_{f_a^i \geq x} f_a^i (\pi_i - \pi_{i-1})$$

The first of these functions is simply a straight line through the origin with slope one. The second function is nonincreasing with x. Initially, when x is zero, all virtual fares are admitted, and all possible terms appear in the summation; then, as x increases, terms drop out as their corresponding virtual fares are exceeded by x. This gives rise to a descending staircase behavior as shown in FIG. 3. Two different cases can occur for the intersection point: either it lies on a horizontal portion of the staircase or it lies on a vertical portion of the staircase.

The intersection point is found by proceeding up the staircase from right to left. As the staircase progress to subsequent virtual fares, first a check is made for a horizontal intersection with the diagonal line, and then the next term is added, checking for a vertical intersection (see FIG. 4). In the case of a vertical intersection, the lowest included virtual fare class is said to be marginal and the EMSR is said to be fixed to the virtual fare. It is almost certain that some of the demand from this marginal fare class will have to be rejected, and the EMSR of a fixed leg may have to be updated more frequently to prevent excessive booking of a marginal demand.

As mentioned above, this nested leg-based method is extended to a network-based method through the mechanism of EMSR-prorated virtual fares. In the case of EMSR-differential virtual fares, the virtual fares on leg a do not change with a change in λ_a alone. However, with EMSR-pro-rated virtual fares, changes in λ_a do have an immediate impact on the virtual fares on leg a. Hence, the iterative structure of FIG. 2 must be modified to allow the effect of this change to be reflected as indicated in FIG. 4.

V. Implementation Of Common Initialization Routines

FIGS. 5–9 are flow charts describing the operation of common initialization routines in a computer program implementing each of the embodiments discussed infra.

The following symbols are used in the flow charts and the descriptions thereof:

1. LEG is an index to a leg of an itinerary or path.
2. ITIN is an itinerary (also termed "path").
3. ITIN-F is an itinerary/fare class combination.
4. LAMBDA(LEG) is the expected marginal seat revenue (EMSR) generated by increasing the capacity of the leg by 1 seat.
5. MEAN(ITIN-F) is the mean of the demand for the ITIN-F.
6. STDDEV(ITIN-F) is the standard deviation of the demand for the ITIN-F (i.e., STDDEV(ITIN-F)^2 is the variance of the demand for the ITIN-F).
7. FARE(ITIN-F) is the fare for an ITIN-F.
8. CAPACITY(LEG) is the seating capacity of the LEG.
9. CURRENT-BUCKETS is a list of ITIN-F's that use a LEG. Each ITIN-F containing the LEG is represented by a bucket element. In this manner, all ITIN-F's

using the LEG can be tracked so that seats can be allocated among the ITIN-F s.

10. Each element of a bucket list contains the following variables:

a. PATH-NUM is the path, i.e., ITIN, that gives rise to the bucket.

b. FARE-IND is the fare class that gives rise to the bucket. Combined with the PATH-NUM this provides the ITIN-F.

c. ELEMENT.MEAN is the mean of the demand for the LEG.

d. ELEMENT.STDDEV is the standard deviation of the demand for the LEG.

e. ELEMENT.TRUE-FARE is the true fare for the ITIN-F.

f. ELEMENT.VIRT-FARE is the virtual fare for the ITIN-F.

g. ELEMENT.LAMBDA-OFFSET is the maximum LAMBDA value, i.e., EMSR, for which the bucket gets seats.

h. ELEMENT.LAMBDA-OTHER is the summation of the LAMBDA(LEG) values for the other LEGs on the ITIN.

i. ELEMENT.SEATS is the number of seats allocated to the ITIN-F.

j. ELEMENT.EXPCFRESQ is a bucket-specific intermediate value.

k. ELEMENT.SQR2PISBF is a bucket-specific constant.

FIG. 5 is a flow chart describing the logic of the GET INPUT routine. The GET INPUT routine inputs data consisting of:

1. The names of the cities served by the airline.

2. The leg information:

a. Flight information;

b. Origin;

c. Destination;

d. Capacity of the carriers;

e. Distance;

3. Itinerary Information:

a. Flight number of the first leg;

b. Origin;

c. Destination;

d. Number of stops;

e. Flight number, origin and destination of each leg component;

f. Fares for each fare class;

g. The μ(ITIN-F), i.e., mean, and σ(ITIN-F), i.e., standard deviation, of demand for each ITIN-F.

Because the present invention may be used with any number of airline reservation systems, the actual operation of the GET INPUT depends on the specific format of the input supplied by the airline reservation system.

FIG. 6 is a flow chart describing the logic of the INIT (INITIALIZATION) routine in the present invention. The INIT routine initializes the global variables of the program, and calculates the starting LAMBDA (LEG) values using a leg-based mileage-prorated EMSR method.

After the INIT routine starts, a first loop is executed once for each path, i.e., itinerary or ITIN, in the network. The total mileage of each ITIN is calculated. The first loop then terminates and a second loop is executed once for each LEG in the network.

If CAPACITY(LEG) is nonzero, then a CONSTRUCT MILE PRORATED FARES routine is called for the LEG depending on the particular virtual fare method (the CONSTRUCT MILE PRORATED FARES routines differ in the details of the data structures each uses). FIG. 7 is a flow chart describing the logic f the CONSTRUCT MILE PRORATED FARES routine in the unnested EMSR-prorated virtual fare method. FIG. 8 is a flow chart describing the logic of the CONSTRUCT MILE PRORATED FARES routine in the nested EMSR-prorated virtual fare method. FIG. 9 is a flow chart describing the logic of the CONSTRUCT MILE PRORATED FARES routine in the nested EMSR-differential virtual fare method. The FARE-SUM and the BUCKET-COUNT is set to zero. A third loop is executed once for each element in the CURRENT-BUCKETS list. The ELEMENT.TRUE-FARE is summed in FARE-SUM and the BUCKET-COUNT is incremented by one. When the third loop terminates, the LAMBDA(LEG) is calculated by dividing the FARE-SUM by two times the BUCKET-COUNT.

If CAPACITY(LEG) is zero, then the LAMBDA(-LEG) is set to a "huge value".

The second loop terminates and the INIT routine terminates.

FIG. 7 is a flow chart describing the logic of the CONSTRUCT MILE PRORATED FARES routine in the unnested EMSR-prorated virtual fare method. The CONSTRUCT MILE PRORATED FARES routine initializes mileage prorated virtual fares to provide suitable initial EMSR's.

The CONSTRUCT MILE PRORATED FARES routine has as its parameter the desired LEG in the network. After the routine starts, all previous buckets are de-allocated. A first loop is executed once for all paths, i.e., ITIN's, in the network containing the LEG. The PRORATION value is calculated by dividing the miles on the LEG by the miles for the ITIN. A second loop is executed for all fare classes in the ITIN. If MEAN(ITIN-F), STDDEV(ITIN-F), and FARE(ITIN-F) are all greater than zero, then LAMBDA-OFFSET is set to zero, TRUE-FARE and VIRT-FARE are set to the value of:

$$(\text{FARE(ITIN-F)})(\text{PRORATION})$$

LAMBDA-OFFSET is set to the value of:

$$(\text{VIRT-FARE})/2(\text{erfc}((-\text{MEAN-(ITIN}-\text{F})/\sqrt{2}(\text{STDDEV(ITIN}-\text{F}))))$$

and SQR2PISBF is set the value of:

$$\sqrt{2\pi}(\text{STDDEV(ITIN}-\text{F}))/(\text{TRUE}-\text{FARE})$$

A new element is created, LAMBDA-OTHER, LAMBDA-OFFSET, MEAN(ITIN-F), STDDEV(ITIN-F), SQR2PISBF, TRUE-FARE, and VIRT-FARE are assigned thereto, and the new element is inserted into the CURRENT-BUCKETS list. The second and first loops terminate, and the CONSTRUCT MILE PRORATED FARES routine terminates.

FIG. 8 is a flow chart describing the logic of the CONSTRUCT MILE PRORATED FARES routine in the nested EMSR-prorated virtual fare method. The CONSTRUCT MILE PRORATED FARES routine initializes mileage prorated virtual fares to provide suitable initial EMSR's.

The CONSTRUCT MILE PRORATED FARES routine has as its parameter the desired LEG in the network. After the routine starts, all previous buckets are de-allocated. A first loop is executed once for all paths, i.e., ITIN's, in the network containing the LEG.

The PRORATION value is calculated by dividing the miles on the LEG by the miles for the ITIN. A second loop is executed for all fare classes in the ITIN. If MEAN(ITIN-F), STDDEV(ITIN-F), and FARE(ITIN-F) are all greater than zero, then TRUE-FARE and VIRT-FARE are set to the value of:

(FARE(ITIN−F))(PRORATION)

A new element is created, LAMBDA-OFFSET, MEAN(ITIN-F), STDDEV(ITIN-F), TRUE-FARE, and VIRT-FARE are assigned thereto, and the new element is inserted into the CURRENT-BUCKETS list. The second and first loops terminate, and the CONSTRUCT MILE PRORATED FARES routine terminates.

FIG. 9 is a flow chart describing the logic of the CONSTRUCT MILE PRORATED FARES routine in the nested EMSR-differential virtual fare method. The CONSTRUCT MILE PRORATED FARES routine initializes mileage prorated virtual fares to provide suitable initial EMSR's.

The CONSTRUCT MILE PRORATED FARES routine has as its parameter the desired LEG in the network. After the routine starts, all previous buckets are de-allocated. A first loop is executed once for all paths, i.e., ITIN's, in the network containing the LEG. The PRORATION value is calculated by dividing the miles on the LEG by the miles for the ITIN. A second loop is executed for all fare classes in the ITIN. If MEAN(ITIN-F), STDDEV(ITIN-F), and FARE(ITIN-F) are all greater than zero, then TRUE-FARE and VIRT-FARE are set to the value of:

(FARE(ITIN−F))(PRORATION)

A new element is created, LAMBDA-OFFSET, MEAN(ITIN-F), STDDEV(ITIN-F), TRUE-FARE, and VIRT-FARE are assigned thereto, and the new element is inserted into the CURRENT-BUCKETS list. The second and first loops terminate, and the CONSTRUCT MILE PRORATED FARES routine terminates.

VI. Implementation Of Unnested EMSR-Prorated Virtual Fare

FIGS. 10–16 are flow charts describing the operation of a number of routines of a computer program implementing the Unnested EMSR-Prorated Virtual Fare Method. For each flight leg in the network which still has residual capacity (i.e., unsold seats), an EMSR-prorated virtual fare is computed for each itinerary/fare class that contains the leg using previously calculated EMSR's. Based on these virtual fares, the new EMSR for the leg is calculated by applying Newton's method to the capacity constraint equation for the leg (which ensures that the number of seats assigned to the various virtual fare classes on the leg is equal to the residual capacity of the flight). If NEWTON's method fails to converge, then a binary search is used. Since each step of Newton's method (or binary search) changes the EMSR of the current leg, the virtual fares must be updated at each step. Once Newton's method (or binary search) converges, then the next leg is processed. The loop terminates when a pass through all legs in the network results in no significant change (by more than a penny) in any EMSR.

The following symbols are used in the flow charts and the descriptions thereof:

1. LEG is an index to a leg of an itinerary or path.
2. ITIN is an itinerary (also termed "path").
3. ITIN-F is an itinerary/fare class combination.
4. LAMBDA(LEG) is the expected marginal seat revenue (EMSR) generated by increasing the capacity of the leg by 1 seat.
5. MEAN(ITIN-F) is the mean of the demand for the ITIN-F.
6. STDDEV(ITIN-F) is the standard deviation of the demand for the ITIN-F (i.e., STDDEV(ITIN-F)^2 is the variance of the demand for the ITIN-F).
7. FARE(ITIN-F) is the fare for an ITIN-F.
8. CAPACITY(LEG) is the mean of the demand for the ITIN-F.
9. DELTA is the change in the EMSR, i.e., the change in LAMBDA (LEG).
10. EPSILON is a tolerance value.
11. CURRENT-BUCKETS is a list of ITIN-F's that use a LEG. Each ITIN-F containing the LEG is represented by a bucket element. In this manner, all ITIN-F's using the LEG can be tracked so that seats can be allocated among the ITIN-F's.
11. FIXED-BUCKETS is a list of elements that have been "fixed", i.e., wherein the virtual fare is approximately equal to the LAMBDA(LEG) value.
12. Each element of a bucket list contains the following variables:
   a. PATH-NUM is the path, i.e., ITIN, that gives rise to the bucket.
   b. FARE-IND is the fare class that gives rise to the bucket. Combined with the PATH-NUM this provides the ITIN-F.
   c. ELEMENT.MEAN is the mean of the demand for the LEG.
   d. ELEMENT.STDDEV is the standard deviation of the demand for the LEG.
   e. ELEMENT.TRUE-FARE is the true fare for the ITIN-F.
   f. ELEMENT.VIRT-FARE is the virtual fare for the ITIN-F.
   g. ELEMENT.LAMBDA-OFFSET is the maximum LAMBDA value, i.e., EMSR, for which the bucket gets seats. This value is required for those demand models, e.g., the Gaussian demand model, that anomalously assign non-zero probability to negative demand. Demand models, e.g., the incomplete gamma distribution, that do not assign non-zero probability to negative demand will not use this value, e.g., ELEMENT.LAMBDA-OFFSET=ELEMENT.VIRT-FARE.
   h. ELEMENT.LAMBDA-OTHER is the summation of the LAMBDA(LEG) values for the other LEGs on the ITIN.
   i. ELEMENT.SEATS is the number of seats allocated to the ITIN-F.
   j. ELEMENT.EXPCFRESQ is a bucket-specific intermediate value.
   k. ELEMENT.SQR2PISBF is a bucket-specific constant.

FIG. 10 is a flow chart describing the MAIN routine in the Unnested EMSR-Prorated Virtual Fare Method. At the start of the MAIN routine, the GET INPUT and INIT routines of FIGS. 6 and 7 respectively are called. The variable DONE is set to the value 1. A first loop is performed for each leg in the network. If CAPACITY(LEG) is greater than zero, then the CONSTRUCT

VIRTUAL FARES routine is called for the leg. The CALC LAMBDA routine is called and if the return value therefrom is 0, then DONE is set to 0. If the CAPACITY(LEG) is zero, then LAMBDA (LEG) is set to a large initial value. When the first loop terminates, after examining each leg in the network, the DONE variable is examined. If DONE is 0, then the loop is performed again; otherwise, the MAIN routine terminates.

FIGS. 11A and 11B describe the CALC LAMBDA routine. The CALC LAMBDA routine is called with a LEG parameter and calculates the LAMBDA (LEG) value. If the CURRENT-BUCKETS list is empty, then the aggregated demand for the leg is far less than the residual capacity of the leg, so LAMBDA (LEG) is set to 0 and a true value is returned to the calling procedure. If the CURRENT-BUCKETS list has elements, then the CALC LIMITS routine is called to calculate the capacity slack, which is stored in the variable G. If G is positive or equal to 0, then the upper bound ULL is set to the value of LAMBDA(LEG) and the lower bound LLL is set to 0. If the variable G is negative, then ULL and LLL are both set to the LAMBDA (LEG) value. A first loop is performed for every element in the CURRENT-BUCKETS list. If ELEMENT.VIRT-FARE is greater than ULL, then ULL is set to the value of ELEMENT.VIRT-FARE. Thus, ULL is set to the maximum virtual fare. When the first loop terminates, if the absolute value of the capacity slack G is less than the value of CAP-TOLER, then the variable CAP-WAS-OK is set to true, otherwise it is set to false, and the routine terminates.

FIG. 12 is a flow chart describing the CALC LIMITS routine in the Unnested EMSR-Prorated Virtual Fare Method. This routine is called with a LEG parameter and calculates the residual capacity for the leg. It also calculates the booking limits for each itinerary/fare class combination. On each itinerary/fare class combination, a number of different special situations can occur: (1) the LAMBDA (LEG) may be less than EPSILON which means that the aggregated demand on the leg is far below the residual capacity and therefore every request can be satisfied; (2) the LAMBDA (LEG) may be greater than the ELEMENT.LAMBDA-OFFSET for the itinerary/fare class combination, in which case the booking limit is set to 0; and (3) the LAMBDA (LEG) may be close to the ELEMENT-.LAMBDA-OFFSET for the itinerary/fare class combination, in which case the itinerary/fare class is stored in a FIXED-BUCKET list.

At the start of the CALC LIMITS routine, the variable G stores the value of CAPACITY(LEG). If LAMBDA (LEG) is less than EPSILON, then a first branch occurs. The local variables TOTAL-MEAN and TOTAL-STDDEV are set to 0. A first loop is performed for every element in the CURRENT-BUCKETS list. If ELEMENT.VIRT-FARE is greater than or equal to EPSILON, then ELEMENT.MEAN is added to TOTAL-MEAN and the square of ELEMENT.STDDEV is added to TOTAL-STDDEV. When the first loop terminates, the square root of the TOTAL-STDDEV is calculated and stored therein. The TOTAL-MEAN is added to three times TOTAL-STDDEV and subtracted from G. If G is greater than 0, then G is set to 0. The routine then terminates. If LAMBDA (LEG) is greater than or equal to EPSILON, then a second branch occurs. The local variables TOTAL-MEAN and TOTAL-STDDEV are set to 0. A FIXED-BUCKETS list is initialized with 0 elements on the list. A second loop is performed for every element in the CURRENT-BUCKETS list. LAMBDA-OFFSET is set to the value of ELEMENT.LAMBDA-OFFSET. If LAMBDA-OFFSET is less than the result of LAMBDA (LEG) minus LAMBDA-RESOLUTION, then ELEMENT.SEAT is set to 0, ELEMENT.EXPCFRESQ is set 0, and the second loop terminates. If LAMBDA-OFFSET is greater than or equal to the result of LAMBDA (LEG) minus LAMBDA-RESOLUTION and LAMBDA-OFFSET is less than or equal to the result of LAMBDA (LEG) plus LAMBDA-RESOLUTION, then ELEMENT.MEAN is added to TOTAL-MEAN; ELEMENT.STDDEV is added to TOTAL-STDDEV; a new element is allocated and inserted into the FIXED-BUCKETS list; ELEMENT.EXPCFRESQ is set to 0, and the second loop terminates. If LAMBDA-OFFSET is greater than the result of LAMBDA (LEG) plus LAMBDA-RESOLUTION, then the local variable DISPLACE is set to the value returned by the CFRE routine, ELEMENT-.SEATS is set to the result of ELEMENT.MEAN plus the result of the square root of 2 multiplied by ELEMENT.STDDEV multiplied by DISPLACE, G has subtracted from it ELEMENT.SEATS, ELEMENT-.EXPCFRESQ is set to the value of the exponential value of DISPLACE squared, and the second loop terminates. After the second loop terminates, then if the variable G is greater than 0, then DISPLACE is set to the result of G minus TOTAL-MEAN, the result thereof divided by TOTAL-STDDEV. If DISPLACE is greater than -2.5, then DISPLACE is set to -2.5. A third loop is performed for all elements in the FIXED-BUCKETS list. The value of ELEMENT.SEATS is set to the result of ELEMENT.MEAN plus the result of ELEMENT.STDDEV multiplied by DISPLACE. The value of G has subtracted from it the value of ELEMENT.SEATS. The first element of the FIXED-BUCKETS list is then deleted and the third loop terminates. On the other hand, if the variable G is less than or equal to 0 after termination of the second loop, then a fourth loop is performed for every element in the FIXED-BUCKETS list. The value of ELEMENT.SEATS is set to 0 and the element is deleted from the FIXED-BUCKETS list.

FIG. 13 is a flow chart describing the CONSTRUCT VIRTUAL FARES routine in the Unnested EMSR-Prorated Virtual Fare Method. This routine constructs EMSR-prorated fares for each bucket element. For each itinerary using the leg, the ELEMENT.LAMBDA-OTHER variable is determined by subtracting the LAMBDA(LEG) from the ELEMENT.LAMBDA-SUM of the itinerary. Then, for each itinerary/fare class combination, the ELEMENT.VIRT-FARE is calculated by multiplying the ELEMENT.TRUE-FARE by the LAMBDA(LEG) and dividing by the LAMBDA-SUM. If the ELEMENT.VIRT-FARE is greater than 0, then a CURRENT-BUCKETS list element is created and linked to the existing CURRENT-BUCKETS list.

The CONSTRUCT VIRTUAL FARES routine is called with a LEG parameter and upon starting, the memory of the previous buckets is de-allocated. A first loop is performed for all paths containing the leg. The value of ELEMENT.LAMBDA-OTHER is set to the value of a negative LAMBDA (LEG). A second loop is performed for every leg of the path. The LAMBDA (LEG) value is added to ELEMENT.LAMBDA-

OTHER. Upon termination of the second loop, a third loop is performed for each fare of the path. If MEAN(ITIN-F), STDDEV(ITIN-F), and FARE(ITIN-F) are all greater than 0, then a new element is allocated and the following actions are performed therefor: the value of ELEMENT.TRUE-FARE is set to the value of FARE(ITIN-F); the value of ELEMENT.VIRT-FARE is set to the result of FARE(ITIN-F) multiplied by LAMBDA(LEG) and divided by the result of LAMBDA(LEG) plus ELEMENT.LAMBDA-OTHER; the value of ELEMENT.LAMBDA-OFFSET is set to the result of:

(ELEMENT.VIRT-FARE)/2
(erfc(−MEAN(ITIN−F)/(√2)
(STDDEV(ITIN−F))))

and the ELEMENT.SQR2PISBF is set to the result of:

(√2π)
(STDDEV(ITIN-F))/(ELEMENT.TRUE-FARE)

FIG. 14 is a flow chart describing the DELFUNC routine. This routine is called with a LEG parameter and calculates the following value for the buckets on a given leg:

$$(\sqrt{2\pi})\,\sigma/(\text{ELEMENT.TRUE-FARE})\,\exp((\text{cfre}(2\lambda/(\text{ELEMENT.VIRT-FARE}))^2)$$

In a simple LAMBDA (LEG) calculation, there is no distinction between true fares and virtual fares, but this routine accounts for the fact that virtual fares change with the LAMBDA (LEG).

At the start of the DELFUNC routine, a local variable SUM is set to 0. A first loop is performed for every element in the CURRENT-BUCKETS list. If ELEMENT.EXPCFRESQ is not 0, then the value of ELEMENT.SQR2PISBF multiplied by ELEMENT-.EXPCFRESQ and the result thereof added to SUM. The value of SUM is returned to the calling routine when DELFUNC terminates.

FIG. 15 is a flow chart describing the UPDATE VIRTUAL FARES routine in the Unnested EMSR-Prorated Virtual Fare Method. This routine is called with a LEG parameter and updates the virtual fares using the new LAMBDA (LEG) value. The virtual fares are calculated by multiplying ELEMENT.TRUE-FARE by the LAMBDA (LEG) and dividing by the LAMBDA-SUM.

At the start of the UPDATE VIRTUAL FARES routine, a first loop is performed for every element in the CURRENT-BUCKETS list. The ELEMENT-.VIRT-FARE is set to the result of ELEMENT.-TRUE-FARE multiplied by LAMBDA(LEG) and divided by the result of LAMBDA(LEG) plus ELEMENT.LAMBDA-OTHER. The ELEMENT-.LAMBDA-OFFSET is set to the result of:

(ELEMENT.VIRT−FARE)/2(erfc(-
(−ELEMENT.MEAN)/(√2)
(ELEMENT.STDDEV))

After the first loop terminates, the routine terminates.

FIG. 16 is a flow chart describing the BINARY routine. This routine is called with a LEG parameter and uses a binary search to calculate LAMBDA(LEG) for a given leg, when the other methods have failed. After the BINARY routine starts (556), a first loop is executed until the difference between ULL and LLL is less than the EPSILON tolerance value, and the absolute value of G is less than the EPSILON tolerance value (558). The LAMBDA(LEG) is calculated as (560):

$$\lambda(LEG) = \frac{ULL + LLL}{2}$$

If G, which is the returned value from the routine CALCULATE LIMITS, is less than 0 (562), then ULL is set to the λ(LEG) value (564). Otherwise (562), if G is greater than or equal to 0 (562), then LLL is set to the λ(LEG) value (566). The first loop terminates when complete (558), and the BINARY routine then terminates (568).

VII. Implementation Of Nested EMSR-Prorated Virtual Fare

FIGS. 17-21 are flow charts describing the operation of a number of routines of a computer program in the Nested EMSR-Prorated Virtual Fare Method. For each flight leg in the network which still has residual capacity, an EMSR-prorated virtual fare is computed for each itinerary/fare class that contains the leg using previously calculated EMSR's. The itinerary/fare classes are then sorted in order of descending virtual fare. Next, by processing this sorted list of virtual fares one-by-one, a trace is made upward to the left along the staircase graph of FIG. 3 to find its intersection with the straight line through the origin. This intersection point defines a new EMSR for the leg, but since EMSR-prorated virtual fares depend on the EMSR of this leg, they must be recalculated and re-sorted, and the intersection point for the leg must be recomputed. This process is repeated until the EMSR on this leg converges; then the next leg is processed. The loop terminates when a pass through all legs in the network results in no significant change (by more than a penny) in any EMSR.

The following symbols are used in the flow charts and the descriptions thereof:

1. LEG is an index to a leg of an itinerary or path.
2. ITIN is an itinerary (also termed "path").
3. ITIN-F is an itinerary/fare class combination.
4. CAPACITY(LEG) is the seating capacity of the LEG.
5. LAMBDA(LEG) is the expected marginal seat revenue (EMSR) generated by increasing the capacity of the leg by 1 seat.
6. MEAN(ITIN-F) is the mean of the demand for the ITIN-F.
7. STDDEV(ITIN-F) is the standard deviation of the demand for the ITIN-F (i.e., STDDEV(ITIN-F)^2 is the variance of the demand for the ITIN-F).
8. FARE(ITIN-F) is the fare for an ITIN-F.
9. DELTA is the change in the EMSR, i.e., LAMBDA (LEG).
10. EPSILON is a tolerance value.
11. FIXED(LEG) is a flag that indicates whether the LEG has been "fixed", i.e., wherein some virtual fare is approximately equal to the LAMBDA(LEG) value.
12. CURRENT-BUCKETS is a list of ITIN-F's that use the LEG. Each ITIN-F containing the LEG is represented by a bucket element. In this manner, all ITIN-F's using the LEG can be tracked so that seats can be allocated among the ITIN-F's.
13. Each element of a bucket list contains the following variables:

a. PATH-NUM is the path, i.e., ITIN, that gives rise to the bucket.
b. FARE-IND is the fare class that gives rise to the bucket.
c. ELEMENT.MEAN is the mean for the bucket.
d. ELEMENT.STDDEV is the standard deviation for the bucket.
e. ELEMENT.TRUE-FARE is the true fare for the corresponding fare class/itinerary combination.
f. ELEMENT.VIRT-FARE is the virtual fare for the bucket.
g. ELEMENT.LAMBDA-OTHER is the summation of LAMBDA(LEG) on other legs in the itinerary.

FIG. 17 is a flow chart describing the MAIN routine in the Nested EMSR-Prorated Virtual Fare Method. At the start of the MAIN routine, the GET INPUT and INIT routines are called. The variable DONE is set to the value 1. A loop is performed for each leg in the network. If the leg has residual capacity, then the CONSTRUCT VIRTUAL FARES routine is called for the leg. The CALC LAMBDA routine is called and if the return value therefrom is 0, then the DONE variable is set to 0. If the leg does not have residual capacity, then LAMBDA (LEG) is set to infinity, which is referred to as HUGE-VAL. When the first loop terminates, after examining each leg in the network, the DONE variable is examined. If the DONE variable is 0, the loop is performed again; otherwise, the MAIN routine terminates.

FIG. 18 is a flow chart describing the CALC LAMBDA routine in the Nested EMSR-Prorated Virtual Fare Method. The CALC LAMBDA routine calculates the LAMBDA (LEG) value. The routine is called with a LEG parameter. If the CURRENT-BUCKETS list is empty, then LAMBDA (LEG) is set to 0 and a value of 1 is returned to the calling procedure. Otherwise, LAMBDA (LEG) is temporarily stored in the variable OLDL. The SORT VIRTUAL FARES routine is called to sort the virtual fares for the leg in ascending order (using any known sort technique). The LAMBDA (LEG) is stored is stored in the variable LASTL and a new LAMBDA (LEG) value is calculated by the FIND INTERSECTION routine. If LAMBDA (LEG) is less than the EPSILON value, then LAMBDA (LEG) is set to 0. The UPDATE VIRTUAL FARES routine is called for the leg. The value of LASTL is subtracted from LAMBDA (LEG), and if the absolute value thereof is greater than or equal to LAMBDA-RESOLUTION, and if LAMBDA (LEG) is greater than EPSILON, then a value of 1 is returned to the calling procedure. Otherwise, the LAMBDA (LEG) is calculated again.

FIG. 19 is a flow chart describing the CONSTRUCT VIRTUAL FARES routine in the Nested EMSR-Prorated Virtual Fare Method. This routine constructs EMSR-prorated fares for each bucket element. This routine is called with a LEG parameter. First, the routine de-allocates previous bucket elements. A first loop is performed for all paths containing the leg. The value of the LAMBDA-SUM variable is set to zero. A second loop is performed for each leg in the path. The LAMBDA (LEG) values for each leg in the path are added to LAMBDA-SUM. After the second loop terminates, LAMBDA-OTHER is set to the value of LAMBDA-SUM minus LAMBDA (LEG). A third loop is performed for each fare of the path. If the MEAN(ITIN-F), STDDEV(ITIN-F), and FARE(ITIN-F) are all greater than 0, then the LAMBDA-SUM is examined to see if it is greater than the EPSILON value. If it is greater, then the ELEMENT.VIRT-FARE is set to the value of ELEMENT.TRUE-FARE multiplied by the LAMBDA (LEG) and divided by the LAMBDA-SUM. If the LAMBDA-SUM is less than the EPSILON value, then the ELEMENT.VIRT-FARE is set to the value of ELEMENT.TRUE-FARE. If the ELEMENT.VIRT-FARE is greater than 0, then memory is allocated for another element in the CURRENT-BUCKET list and the element is inserted therein. The third loop and first loops terminate and the routine terminates.

FIG. 20 is a flow chart describing the FIND INTERSECTION routine in the Nested EMSR-Prorated Virtual Fare Method. This routine is called with a LEG parameter. At the start, local variables NEW-LEVEL, HIGHER-REJ, CUM-MEAN, AND CUM-VAR are set to 0. A first loop is performed for every element in the CURRENT-BUCKETS list so long as ELEMENT.VIRT-FARE for each elements is greater than NEW-LEVEL. Each ELEMENT.MEAN is added to CUM-MEAN. Each ELEMENT.STDDEV is squared and added to CUM-VAR. A value for the variable PROB is calculated as:

$$PROB = erfc\left(\frac{1}{2}\left(\frac{(CAPACITY(LEG)) - (CUM\text{-}MEAN)}{\sqrt{2\,(CUM\text{-}VAR)}}\right)\right)$$

The variable PROB could be calculated differently if an alternate demand distribution model is used.

ELEMENT.VIRT-FARE is multiplied by the result of PROB minus HIGHER-REJ, and the result added to NEW-LEVEL. Then, HIGHER-REJ is set equal to PROB. Upon termination of the first loop, the ELEMENT.VIRT-FARE of the last element processed in the first loop is compared to NEW-LEVEL. If ELEMENT.VIRT-FARE of the last element is less than NEW-LEVEL, then NEW-LEVEL is set to the value of ELEMENT.VIRT-FARE of the last element and the FIXED(LEG) is set to a value of 1. If the ELEMENT.VIRT-FARE of the last element is greater than or equal to NEW-LEVEL, then FIXED(LEG) is set to a value of 0. The NEW-LEVEL value is then returned to the calling routine.

FIG. 21 is a flow chart describing the UPDATE VIRTUAL FARES routine in the Nested EMSR-Prorated Virtual Fare Method. This routine is called with a LEG parameter. A local variable PREV-FARE is set to infinity, which is referred to as HUGE-VAL. A first loop is performed for all elements in the CURRENT-BUCKETS list. For each element, the LAMBDA(LEG) is added to ELEMENT.LAMBDA-OTHER, and the result therefrom is stored in LAMBDA-SUM. If LAMBDA-SUM is greater than or equal to EPSILON, then ELEMENT.VIRT-FARE is set to the value of the ELEMENT.TRUE-FARE multiplied by LAMBDA (LEG) and divided by LAMBDA-SUM. If LAMBDA-SUM is less than EPSILON, then ELEMENT.VIRT-FARE is set to the value of ELEMENT.TRUE-FARE. If ELEMENT.VIRT-FARE is greater than the PREV-FARE, then the element is advanced through the CURRENT-BUCKETS list until an ordering of decreasing ELEMENT.VIRT-FARES is re-established. If ELEMENT.VIRT-FARE

is less than or equal to PREV-FARE, then PREV-FARE is set to the value of ELEMENT.VIRT-FARE.

VIII. Implementation Of Nested EMSR-Differential Virtual Fare

FIGS. 22-25 are flow charts describing the operation of a number of routines of a computer program implementing the Nested EMSR-Differential Virtual Fare Method. For each flight leg in the network which still has residual capacity, an EMSR-differential virtual fare is computed for each itinerary/fare class that contains the leg using previously calculated EMSR's. The itinerary/fare classes are then sorted in order of descending virtual fare. Next, by processing this sorted list of virtual fares one-by-one, a trace is made upward to the left along the staircase graph of FIG. 3 to find its intersection with the straight line through the origin. This intersection point defines the new EMSR for the leg. The next leg is then processed. The loop terminates when a pass through all legs in the network results in no significant change (by more than a penny) in any EMSR.

The following symbols are used in the flow charts and the descriptions thereof:

1. LEG is an index to a leg of an itinerary or path.
2. ITIN is an itinerary (also termed "path").
3. ITIN-F is an itinerary/fare class combination.
4. CAPACITY(LEG) is the mean of the demand for the ITIN-F.
5. LAMBDA(LEG) is the expected marginal seat revenue (EMSR) generated by increasing the capacity of the leg by 1 seat.
6. FIXED(LEG) is a flag that indicates whether the LEG has been "fixed", i.e., wherein some virtual fare is approximately equal to the LAMBDA(LEG) value.
7. MEAN(ITIN-F) is the mean of the demand for the ITIN-F.
8. STDDEV(ITIN-F) is the standard deviation of the demand for the ITIN-F (i.e., STDDEV(ITIN-F)^2 is the variance of the demand for the ITIN-F).
9. FARE(ITIN-F) is the fare for an ITIN-F.
10. DELTA is the change in the EMSR, i.e., LAMBDA (LEG).
11. EPSILON is a tolerance value.
12. CURRENT-BUCKETS is a list of ITIN-F's that use a LEG. Each ITIN-F containing the LEG is represented by a bucket element. In this manner, all ITIN-F's using the LEG can be tracked so that seats can be allocated among the ITIN-F's.
13. Each element of a bucket list contains the following variables:
    a. PATH-NUM is the path index that gives rise to the bucket.
    b. FARE-IND is the fare class index that gives rise to the bucket.
    c. ELEMENT.MEAN is the mean of the demand for the ITIN-F.
    d. ELEMENT.STDDEV is the standard deviation of the demand for the ITIN-F.
    e. ELEMENT.TRUE-FARE is the true fare for the corresponding fare class/itinerary combination.
    f. ELEMENT.VIRT-FARE is the virtual fare for the bucket.

FIG. 22 is a flow chart describing the MAIN routine in the Nested EMSR-Differential Virtual Fare Method. At the start of the MAIN routine, the GET INPUT and INIT routines are called. The variable DONE is set to the value 1. A first loop is performed for each leg in the network. If the leg has residual capacity, then the CONSTRUCT VIRTUAL FARES routine is called for the leg. The CALC LAMBDA routine is called and the return value therefrom is saved. If the return value is 0, then DONE is set to 0. If the leg does not have residual capacity, then LAMBDA (LEG) is set to infinity, which is referred to as HUGE-VAL. When the first loop terminates, after examining each leg in the network, the DONE variable is examined. If DONE is 0, then the loop is performed again; otherwise, the MAIN routine terminates.

FIG. 23 is a flow chart describing the CALC LAMBDA routine in the Nested EMSR-Differential Virtual Fare Method. The CALC LAMBDA routine is called with a LEG parameter and calculates the LAMBDA (LEG) value. At the start of the routine, if the CURRENT-BUCKETS list is empty, then LAMBDA (LEG) is set to 0 and a true value is returned to the calling routine. Otherwise, the variable OLDL is set to the value of LAMBDA (LEG). The SORT VIRTUAL FARE routine is called for the leg. The FIND INTERSECTION routine is called for the leg and the value returned therefrom is stored in the LAMBDA (LEG). If the absolute value of the results of subtracting OLDL from LAMBDA (LEG) is less than EPSILON, then a true value is returned to the calling routine; otherwise a false value is returned.

FIG. 24 is a flow chart describing the CONSTRUCT VIRTUAL FARES routine in the Nested EMSR-Differential Virtual Fare Method. This routine is called with a LEG parameter and constructs EMSR-prorated fares for each bucket element. For each itinerary using the leg, the ELEMENT.LAMBDA-OTHER variable is determined by subtracting the LAMBDA(LEG) from the ELEMENT.LAMBDA-SUM of the itinerary. Then, for each itinerary/fare class combination, the ELEMENT.VIRT-FARE is calculated by subtracting ELEMENT.LAMBDA-OTHER from the ELEMENT.TRUE-FARE. If the ELEMENT.VIRT-FARE is greater than 0, then a CURRENT-BUCKETS list element is created and linked to the existing CURRENT-BUCKETS list.

The CONSTRUCT VIRTUAL FARES routine is called with a LEG parameter and constructs the ELEMENT.VIRT-FARE values. At the start of the routine, the memory of previous buckets is deallocated. A first loop is performed for all paths containing the leg. The variable LAMBDA-OTHER is set to the negative value of LAMBDA (LEG). A second loop is performed for all legs in the path. For each leg in the path, the LAMBDA (LEG) value is added to the LAMBDA-OTHER value. Upon termination of the second loop, a third loop is performed for each fare in the path. If the MEAN(ITIN-F), STDDEV(ITIN-F), and FARE(ITIN-F) are all greater than 0, then ELEMENT.TRUE-FARE is set to the value of FARE(ITIN-F) and ELEMENT.VIRT-FARE is set to the result of FARE(ITIN-F) minus LAMBDA-OTHER. If ELEMENT.VIRT-FARE is greater than 0, then a new element is created for the CURRENT-BUCKETS list and inserted therein.

FIG. 25 is a flow chart describing the FIND INTERSECTION routine in the Nested EMSR-Differential Virtual Fare Method. This routine is called with a LEG parameter. A number of local variables are initialized to 0, including NEW-LEVEL, HIGHER-REJ, CUM-MEAN, and CUM-VAR. A first loop is performed for all elements in the CURRENT-BUCKETS list while

each ELEMENT.VIRT-FARE is greater than the value of NEW-LEVEL. The ELEMENT.MEAN is accumulated in the variable CUM-MEAN and the ELEMENT.STDDEV is squared and added to CUM-VAR. The variable PROB is calculated as:

$$\frac{1}{2}\text{erfc}((\text{CAPACITY(LEG)}-(\text{CUM}-\text{MEAN}))/\sqrt{2}(\text{CUM}-\text{VAR}))$$

The value of ELEMENT.VIRT-FARE is multiplied by the result of PROB minus HIGHER-REJ and the result thereof added to NEW-LEVEL. The value of HIGHER-REJ is set to the value of PROB. Upon termination of the first loop, if the last ELEMENT.VIRT-FARE is less than NEW-LEVEL, then NEW-LEVEL is set to the value of ELEMENT.VERT-FARE and FIXED(LEG) is set to 1. If the last ELEMENT.VIRT-FARE is less than NEW-LEVEL, then FIXED(LEG) is set to 0. The value of NEW-LEVEL is returned to the calling routine.

IX. Conclusion

This concludes the description of the three preferred embodiments of the present invention. The following paragraphs describe some alternative embodiments of the invention.

Any number of other yield management solutions could be substituted for the airline seat inventory control system described herein. Instead of using seats, itinerary/fare classes, and flight legs, the invention could be formulated in terms of resources, demand categories, and resource categories, respectively. This would allow the invention to be generalized to other embodiments, but would not entail any revision to the solution offered. It is believed that most yield management systems can be described in the general terms given above.

In summary, an airline seat reservation system has been described wherein seat reservations are controlled using, in part, a seat inventory control system. The seat reservation control, based on a concept termed Network-Based Expected Marginal Seat Revenue (EMSR), does not require the large number of variables required by the other network-based approaches, and it incorporates a probabilistic demand model without resorting to computationally intractable integer programming. The computer-based seat inventory control system uses leg-based methods to control bookings in a flight network comprised of a plurality of itinerary/fare class combinations using a plurality of flight legs. When considering a particular flight leg, the total fare paid by a using the leg is discounted by taking into account the displacement cost of the travel on the other legs of the itinerary to create a virtual fare. Expected marginal seat revenues provide the displacement cost on the legs when computing virtual fares. Using these EMSR-dependent virtual fares, a leg-based optimization method is applied to the virtual fares to obtain network-optimal results.

The foregoing descriptions of the preferred embodiments of the invention have been presented for the purposes of illustration and description. These preferred embodiments are not intended to be exhaustive nor to limit the invention to the precise forms disclosed. Many modifications and variations are possible in light of the above teaching. It is intended that the scope of the invention be limited not by this detailed description, but rather by the claims appended thereto.

TABLE 1

	Comparison Of Seat Allocation Methods			
Method	Optimal Basis	Demand Characteristics	Number Of Variables	Computational Complexity
Leg-based EMSR	Leg	Stochastic	Small	Linear
Virtual-Nesting EMSR	Leg	Stochastic	Small	Linear
Deterministic LP	Network	Deterministic	Large	Large Polynomial
Probabilistic LP	Network	Stochastic	Very Large	Exponential
Network-Based EMSR	Network	Stochastic	Small	Small Polynomial

What is claimed is:

1. An airline seat reservation system, comprising:

(a) a programmed computer;

(b) data storage means, connected to the programmed computer, for storing a database describing a flight network and seat reservation requests, the flight network comprising a plurality of flight legs a, and itinerary p and fare class i combinations, each flight leg a having a residual seating capacity C_a, and each itinerary p and fare class i combination having a revenue yield f^i_p for a seat reserved therein;

(c) the programmed computer comprising seat assignment means for processing the database describing the flight network to assign seats in a flight leg a to one or more itinerary p and fare class i combinations, the seat assignment means comprising:

(1) means for calculating an initial expected marginal seat revenue (EMSR) λ_a for all flight legs a;

(2) means for computing an unnested EMSR-prorated virtual fare $v^i_{p,a}$ for every itinerary p and fare class i combination that contains a particular flight leg a having a nonzero residual seating capacity C_a so that:

$$v^i_{p,a} \triangleq \frac{f^i_p \lambda_a}{\sum_{b \in p} \lambda_b}$$

(3) means for calculating a new EMSR λ_a for the particular flight leg a based on the virtual fares $v^i_{p,a}$ by applying Newton's method to a seating capacity constraint for the particular flight leg a:

$$\sum_{\substack{p \\ a \in p \, v^i_{p,a} > \lambda_a}} \sum_i Q_{p^i}(\lambda_a / v^i_{p,a}) = C_a$$

wherein $Q^i{}_p$ is an inverse of a cumulative probability density function of demand for fare class i travel on itinerary p, thereby ensuring that a total number of seats assigned to the itinerary p and fare class i combinations are equal to the residual seating capacity of the particular flight leg a, wherein the virtual fares $v^i_{p,a}$ are updated at each step of the

Newton's method since each step changes the EMSR λ_a for the particular flight leg a;

(4) means for converging the EMSR λ_a for the particular flight leg a;

(5) means for terminating the seat assignment means when the changes in the EMSR's λ_a's for all flight legs a are insignificant;

(d) reservation terminal means, operatively connected to the programmed computer, for entering a seat reservation request for a particular itinerary p and fare class i combination; and

(e) the programmed computer comprising reservation means for receiving the seat reservation request for the particular itinerary p and fare class i combination from the reservation terminal means, for accepting the seat reservation request in accordance with a value selected from a group comprising at least one of the following: a globally optimal set of EMSR's λ_a's and the total number of seats assigned to the itinerary pa nd fare class i combinations for recording the seat reservation request in the database, and for transmitting an electronic status indication of the seat reservation request from the computer to the reservation terminal means.

2. The apparatus of claim 1, wherein the means for calculating further comprises means for performing a binary search when the EMSR λ_a fails to converge with Newton's method.

3. The apparatus of claim 1, wherein the means for terminating further comprises means for terminating the seat assignment means when the changes in EMSR's λ_a's for all flight legs a are less than a tolerance value.

4. The apparatus of claim 1, wherein the reservation means further comprises means for accepting the seat reservation request when the total number of seats assigned to the itinerary p and fare class i combinations is not exceeded.

5. The apparatus of claim 1, wherein the reservation means further comprises means for accepting the seat reservation request when the seat reservation request would yield revenue greater than or equal to a sum of the EMSR's λ_a's for all flight legs a in the itinerary p.

6. An airline seat reservation system, comprising:

(a) a programmed computer;

(b) data storage means, connected to the programmed computer, for storing a database describing a flight network and seat reservation requests, the flight network comprising a plurality of flight legs a, and itinerary p and fare class i combinations, each flight leg a having a residual seating capacity C_a, and each itinerary p and fare class i combination having a revenue yield f^i_p for a seat reserved therein;

(c) the programmed computer comprising seat assignment means for processing the database describing the flight network to assign seats in a flight leg a to one or more itinerary p and fare class i combinations, the seat assignment means comprising:

(i) means for calculating an initial expected marginal sat revenue (EMSR) λ_a for all the flight legs a;

(2) means for computering a nested EMSR-prorated virtual fare $v^i_{p,a}$ for each itinerary p and fare class i combination that contains a particular flight leg a having a nonzero residual seating capacity C_a so that:

$$v^i_{p,a} \triangleq \frac{f^i_p \lambda_a}{\sum_{b \in p} \lambda_b}$$

(3) means for sorting the itinerary p and fare class i combinations into a list ordered by descending values of virtual fares $v^i_{p,a}$;

(4) means for processing the sorted list of virtual fares $v^i_{p,a}$ one-by-one to find an intersection point defining a new EMSR λ_a for the particular flight leg a between functions:

$$\lambda_a = x$$

$$\lambda_a = \sum_{f^j_a \geq x} f^j_a (\pi_j - \pi_{j-1})$$

wherein x is an independent variable, f^j_a is a jth largest virtual fare on leg a, and π_j is a probability that more than C_a passengers are willing to pay f^j_a or more to travel on leg a;

(5) means for converging the EMSR λ_a for the particular flight leg a;

(6) means for terminating the seat assignment means when the changes in the EMSR's λ_a's for all flight legs a are insignificant;

(d) reservation terminal means, operatively connected to the programmed computer, for entering a seat reservation request for a particular itinerary p and fare class i combination; and

(e) the programmed computer comprising reservation means for receiving the seat reservation request for the particular itinerary p and fare class i combination from the reservation terminal means, for accepting the seat reservation request in accordance with a globally optimal set of EMSR's λ_a's, for recording the seat reservation request in the database, and for transmitting an electronic status indication of the seat reservation request from the computer to the reservation terminal means.

7. The apparatus of claim 6, wherein the means for terminating further comprises means for terminating the seat assignment means when the changes in EMSR's λ_a's for all flight legs a are less than a tolerance value.

8. The apparatus of claim 6, wherein the reservation means further comprises means for accepting the seat reservation request when the seat reservation request would yield revenue greater than or equal to a sum of the EMSR's λ_a's for all flight legs a in the itinerary p.

9. An airline seat reservation system, comprising:

(a) a programmed computer;

(b) data storage means, connected to the programmed computer, for storing a database describing a flight network and seat reservation requests, the flight network comprising a plurality of flight legs a, and itinerary p and fare class i combinations, each flight leg a having a residual seating capacity C_a, and each itinerary p and fare class i combination having a revenue yield f^i_p for a seat reserved therein;

(c) the programmed control comprising seat assignment means for processing the database describing the flight network to assign seats in a flight leg a to one or more itinerary p and fare class i combinations, the seat assignment means comprising:

(1) means for calculating an initial expected marginal seat revenue (EMSR) λ_a for all the flight legs a;

(2) means for computing a nested EMSR-differential virtual fare $v^i_{p,a}$ for each itinerary p and fare class i combination that contains a particular flight leg a having a nonzero residual seating capacity C_a so that:

$$v^i_{p,a} \triangleq f^i_p - \sum_{\substack{b \in p \\ b \neq a}} \lambda_b$$

(3) means for sorting the itinerary p and fare class i combinations into a list ordered by descending virtual fares $v^i_{p,a}$ for the particular flight leg a;

(4) means for processing the sorted list of virtual fares $v^i_{p,a}$ one-by-one to find an intersection point defining a new EMSR λ_a for the particular flight leg a between functions:

$$\lambda_a = x$$

$$\lambda_a = \sum_{f^j_d \geq x} f^j_d (\pi_j - \pi_{j-1})$$

wherein x is an independent variable, f^j_a is a jth largest virtual fare on leg a, and π_j is a probability that more than C_a passengers are willing to pay the virtual fare f^j_a or more to travel on leg a;

(5) means for terminating the seat assignment means when the changes in the EMSR's λ_a's for all flight legs a are insignificant;

(d) reservation terminal means, operatively connected to the programmed computer, for entering a seat reservation request for a particular itinerary p and fare class i combination; and

(e) the programmed computer comprising reservation means for receiving the seat reservation request for the particular itinerary p and fare class i combination from the reservation terminal means, for accepting the seat reservation request in accordance with a globally optimal set of EMSR's λ_a's, for recording the seat reservation request in the database, and for transmitting an electronic status indication of the seat reservation request to the reservation terminal means.

10. The apparatus of claim **9**, wherein the means for terminating further comprises means for terminating the seat assignment means when the changes in EMSR's λ_a's for all flight legs a are less than a tolerance value.

11. The apparatus of claim **9**, wherein the reservation means further comprises means for accepting the seat reservation request when the seat reservation request would yield revenue greater than or equal to a sum of the EMSR's λ_a s for all flight legs a in the itinerary p.

12. A system for allocating physical resources, comprising:

(a) a computer;

(b) data storage means, connected to the computer, for storing a database describing a known resource capacity for each of a plurality of resource categories a, a known demand distribution for each of a plurality of demand categories z, a known revenue yield for a resource reserved within each demand category z, and a booking limit for each demand category z;

(c) the computer comprising resource assignment means for processing the database to assign resources in a resource category a to ne r more demand categories z, the resource assignment means comprising:

(1) means for calculating an initial expected marginal resource revenue (EMRR) λ_a for all resource categories a;

(2) means for computing an unnested EMRR-prorated virtual price $v_{z,a}$ for every demand category z that contains a particular resource category a having a nonzero residual capacity C_a so that:

$$v_{z,a} \triangleq \frac{f_z \lambda_a}{\sum_{b \in z} \lambda_b}$$

(3) means for calculating a new EMRR λ_a for the particular demand category a based on the virtual prices $v_{z,a}$ by applying Newton's method to a resource capacity constraint for the particular resource category a:

$$\sum_{\substack{a \in z \\ v_{z,a} > \lambda_a}} Q_z(\lambda_a / v_{z,a}) = C_a$$

wherein Q_z is an inverse of a cumulative probability density function of demand for demand category z, thereby ensuring that a total number of resources assigned to the demand category z are equal to the residual capacity C_a of the particular resource category a, wherein the virtual prices $v_{z,a}$ are updated at each step of the Newton's method since each step changes the EMRR λ_a for the particular resource category a;

(4) means for converging the EMRR λ_a for the particular resource category a;

(5) means for terminating the resource assignment means when the changes in the EMRR's λ_a's for all resource categories a are insignificant;

(d) reservation terminal means, operatively connected to the computer, for entering a resource reservation request for a particular demand category z; and

(e) the computer comprising reservation means, for receiving the resource reservation request for the particular demand category z from the reservation terminal means, for accepting the resource reservation request in accordance with a globally optimal set of EMRR's λ_a's, for recording the resource reservation request in the database, and for transmitting an electronic status indication of the resource reservation request to the reservation terminal means.

13. The apparatus of claim **12**, wherein the means for calculating further comprises means for performing a binary search when the EMRR λ_a fails to converge with Newton's method.

14. The apparatus of claim **12**, wherein the means for terminating further comprises means for terminating the resource assignment means when the changes in EMRR's λ_a's for all resource categories a are less than a tolerance value.

15. The apparatus of claim 12, wherein the reservation means further comprises means for accepting the resource reservati n request when the total number of resources assigned to the demand categories z is not exceeded.

16. The apparatus of claim 12, wherein the reservation means further comprises means for accepting the resource reservation request when the resource reservation request would yield revenue greater than or equal to a sum of the EMRR's λ_a's for all resource categories a in the demand categories z.

17. A system for allocating physical resources, comprising:

(a) a computer;

(b) data storage means, connected to the computer, for storing a database describing a known resource capacity for each of a plurality of resource categories a, a known demand distribution for each of a plurality of demand categories z, a known revenue yield for a resource reserved within each demand category z and a booking limit for each demand category z;

(c) the computer comprising resource assignment means for processing the database to assign resources in a resource category a to one or more demand categories z, the resource assignment means comprising:

(1) means for calculating an initial expected marginal resource revenue (EMRR) λ_a for all the resource categories a;

(2) means for computering a nested EMRR-prorated virtual price $v_{z,a}$ for each demand category z that contains a particular resource category a having a nonzero residual capacity C_a so that:

$$v_{z,a} \triangleq \frac{f_p \lambda_a}{\sum_{b \in z} \lambda_b}$$

(3) means for sorting the demand categories z into a list ordered by descending values of virtual prices $v_{z,a}$;

(4) means for processing the sorted list of virtual prices $v_{z,a}$ one-by-one to find an intersection point defining a new EMRR λ_a for the particular resource category a between functions:

$$\lambda_a = x$$

$$\lambda_a = \sum_{f_a^j \geq x} f_a^j (\pi_j - \pi_{j-1})$$

wherein x is an independent variable, f_a^j is a jth largest virtual price in resource category a, and π_j is a probability that more than C_a customers are willing to pay f_a^j or more for a reservation in resource category a;

(5) means for converging the EMRR λ_a for the particular resource categories a;

(6) means for terminating the resource assignment means when the changes in the EMRR's λ_a's for all resource categories a are insignificant;

(d) reservation terminal means, operatively connected to the computer, for entering a resource reservation request for a particular demand category z; and

(e) the computer comprising reservation means for receiving the resource reservation request for the particular demand category z from the reservation terminal means, for accepting the resource reservation request in accordance with a globally optimal set of EMRR's λ_a's, for recording the resource reservation request in the database, and for transmitting an electronic status indication f the resource reservation request to the reservation terminal means.

18. The apparatus of claim 17, wherein the means for terminating further comprises means for terminating the resource assignment means when the changes in EMRR's λ_a's for all resource categories a are less than a tolerance value.

19. The apparatus of claim 17, wherein the reservation means further comprises means for accepting the resource reservation request when the resource reservation request would yield revenue greater than or equal to a sum of the EMRR's λ_a's for all resource categories a in the demand category z.

20. A system for allocating physical resources, comprising:

(a) a computer;

(b) data storage means, connected to the computer, for storing a database describing a known resource capacity for each of a plurality of resource categories a, a known demand distribution for each of a plurality of demand categories z, a known revenue yield for a resource reserved within each demand category a, and a booking limit for each demand category z;

(c) the computer comprising resource assignment means for processing the database to assign resources in a resource category a to one or more demand categories z, the resource assignment means comprising:

(1) means for calculating an initial expected marginal resource revenue (EMRR) λf_a for all the resource categories a;

(2) means for computing a nested EMRR-differential virtual price $v_{z,a}$ for each demand category z that utilizes a particular resource category a having a nonzero residual capacity C_a so that:

$$v_{z,a} \triangleq f_p - \sum_{\substack{b \in z \\ b \neq a}} \lambda_b$$

(3) means for sorting the demand categories z into a list ordered by descending virtual prices $v_{z,a}$;

(4) means for processing the sorted list of virtual prices $v_{z,a}$ one-by-one to find an intersection point defining a new EMRR λ_a for the particular resource category a between functions:

$$\lambda_a = x$$

$$\lambda_a = \sum_{f_a^j \geq x} f_a^j (\pi_j - \pi_{j-1})$$

wherein x is an independent variable, f_a^j is a jth largest virtual price in resource category a, and π_j is a probability that more than C_a customers are willing to pay the virtual price f_a^j or more for a for a reservation in resource category a;

(5) means for terminating the resource assignment means when the changes in the EMRR's λ_a's for all resource categories a are insignificant;

(d) reservation terminal means, operatively connected to the computer, for entering a resource reservation request for a particular demand category z; and

(e) The computer comprising reservation means for receiving the resource reservation request for the particular demand category z from the reservation terminal means, for accepting the resource reservation request in accordance with a globally optimal set of EMRR's λ_a's, for recording the resource reservation request in the database, and for transmitting an electronic status indication of the resources reservation request to the reservation terminal means.

21. The apparatus of claim 20, wherein the means for terminating further comprises means for terminating the resource assignment means when the changes in EMRR's λ_a's for all resource categories a are less than a tolerance value.

22. The apparatus of claim 20, wherein the reservation means further comprises means for accepting the resource reservation request when the resource reservation request would yield revenue greater than or equal to a sum of the EMRR's λ_a's for all resource categories a in the demand category z.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,270,921

DATED : December 14, 1993

INVENTOR(S) : Scot W. Hornick

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

In column 3, line 20, delete "combined are" after the word --is--.

In column 6, line 4, "$f_a^j \geq x$" should read --$f_a^j \geq x$--.

In column 6, line 36, "$f_a^j \geq x$" should read --$f_a^j \geq x$--.

In column 8, line 43, delete "15" after the word --fare--.

In column 9, line 39, "$\geq$" should read --$\geq$--.

In column 9, line 52, "$\geq$" should read --$\geq$--.

In column 9, line 52, "$E_{i-1})$" should read -- $\overline{E}_{i-1})$--.

In column 9, line 59, "$E_{i-1})$" should read -- $\overline{E}_{i-1})$--.

In column 9, line 60, "$\geq$" should read --$\geq$--.

In column 9, line 67, "$\geq$" should read --$\geq$--.

In column 10, line 8, "$\geq$" should read --$\geq$--.

In column 11, line 2, "F s" should read --F's--.

In column 12, line 46, "√2" should read -- $\sqrt{2}$ --.

In column 17, line 14, "√2" should read -- $\sqrt{2}$ --.

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,270,921

DATED : December 14, 1993

INVENTOR(S) : Scot W. Hornick

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

In column 23, line 7-8, "√2 (CUM-VAR))" should read

-- $\sqrt{2(CUM\text{-}VAR)}$)--.

In column 25, line 20, "pa nd" should read --p and--.

In column 25, line 61, "(i)" should read --(1)--.

In column 25, line 62, "sat" should read --seat--.

In column 25, line 64, "computering" should read --computing--.

In column 26, line 18, "$\geqq$" should read --≥--.

In column 26, line 20, "f_a^j" should read -- f_a^j --.

In column 26, line 23, "j_a" should read -- f_a^j --.

In column 27, line 27, "$\geqq$" should read --≥--.

In column 27, line 29, "f_a^j" should read -- f_a^j --.

In column 27, line 32, "f_a^j" should read -- f_a^j --.

In column 27, line 59, "λ_a s" should read --λ_a's--.

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,270,921
DATED : December 14, 1993
INVENTOR(S) : Scot W. Hornick

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

In column 29, line 53, "$\geq$" should read --$\geq$--.

In column 29, line 55, "f_a^j" should read -- f_a^j --.

In column 29, line 58, "f_a^j" should read -- f_a^j --.

In column 30, line 31, "a" should read --z--.

In column 30, line 40, "λf_a" should read --λ_a--.

In column 30, line 62, "$\geq$" should read --$\geq$--.

In column 30, line 64, "f_a^j" should read -- f_a^j --.

In column 30, line 67, "f_a^j" should read -- f_a^j --.

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,270,921
DATED : December 14, 1993
INVENTOR(S) : Scot W. Hornick

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Column 31, line 8, "The" should read —the—.

Signed and Sealed this

Twenty-ninth Day of November, 1994

Attest:

Bruce Lehman

BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks

US006295521B1

(12) **United States Patent**
DeMarcken et al.

(10) **Patent No.: US 6,295,521 B1**
(45) **Date of Patent: Sep. 25, 2001**

(54) **TRAVEL PLANNING SYSTEM**

(75) Inventors: **Carl G. DeMarcken**, Arlington; **Jeremy Wertheimer**, Brookline, both of MA (US)

(73) Assignee: **ITA Software, Inc.**, Cambridge, MA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/109,327**

(22) Filed: **Jul. 2, 1998**

(51) **Int. Cl.**[7] **G06F 17/60**
(52) **U.S. Cl.** **705/6**; 705/5; 705/400
(58) **Field of Search** 701/201, 202; 705/5, 6, 400; 707/10, 104; 709/217, 218, 219

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,862,357	*	8/1989	Ahlstrom et al.	705/6
5,021,953		6/1991	Webber et al.	364/407
5,177,684	*	1/1993	Harker et al.	701/117
5,191,523		3/1993	Whitesage	364/407
5,237,499	*	8/1993	Garback	705/6
5,253,166	*	10/1993	Dettelbach et al.	705/5
5,255,184	*	10/1993	Hornick et al.	705/6
5,331,546	*	7/1994	Webber et al.	705/6
5,422,809		6/1995	Griffin et al.	364/407
5,570,283		10/1996	Shoolery et al.	364/407
5,623,413	*	4/1997	Matheson et al.	701/117
5,644,721		7/1997	Chung et al.	395/206
5,732,398	*	3/1998	Tagawa	705/6
5,794,172	*	8/1998	Matheson et al.	701/117
5,797,127		8/1998	Walker et al.	705/5
5,832,454	*	11/1998	Jafri et al.	705/6
5,838,973		11/1998	Carpenter-Smith	395/701
5,897,620	*	4/1999	Walker et al.	705/5
5,948,040	*	9/1999	DeLorme et al.	701/201
6,023,679	*	2/2000	Acebo et al.	705/5

FOREIGN PATENT DOCUMENTS

WO 89/07798	* 8/1989	(EP) .
0 451 371 A1	10/1991	(EP) .
0 762 306 A2	3/1997	(EP) .
WO 97/08639	3/1997	(WO) .

OTHER PUBLICATIONS

Poling: "Euram readying CompuServe link. (Euram Tours offers travel agents a data base of its special fares)"; Travel Weekly, v52, n84, Oct. 1993, pp. 1–2.*

"Towards desktop personal travel agents", D.T. Ndumu, J.C. Collis and H.S. Nwana, BT Technol. J. vol. 16, No. 3, Jul. 1998, pp. 69–78.

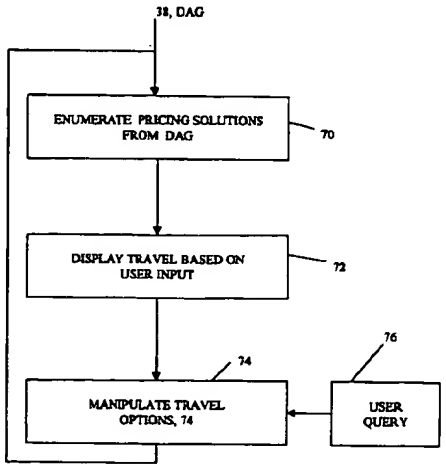
* cited by examiner

Primary Examiner—Edward R. Cosimano
(74) *Attorney, Agent, or Firm*—Fish & Richardson PC

(57) **ABSTRACT**

An airline travel planning system is described. The system includes a server computer executing a server process including a search process to search for set of pricing solutions in accordance with at least one destination and at least one origin. The search process represents the set of pricing solutions in the form of a directed acyclic graph. The system also includes a client computer executing a client process on the set of pricing solutions. The client process has a manipulation process that manipulates the set of pricing solutions in response to user preferences. Several processes are described including a process responsive to user preferences and to set of pricing solutions that provides pricing solutions sorted by said user preference, a process that sorts set of pricing solutions to produce a subset of said set of pricing solutions in accordance with user specified preferences, and a process that prunes from the directed acyclic graph nodes that are no longer contained within the subset of set of pricing solutions.

33 Claims, 33 Drawing Sheets



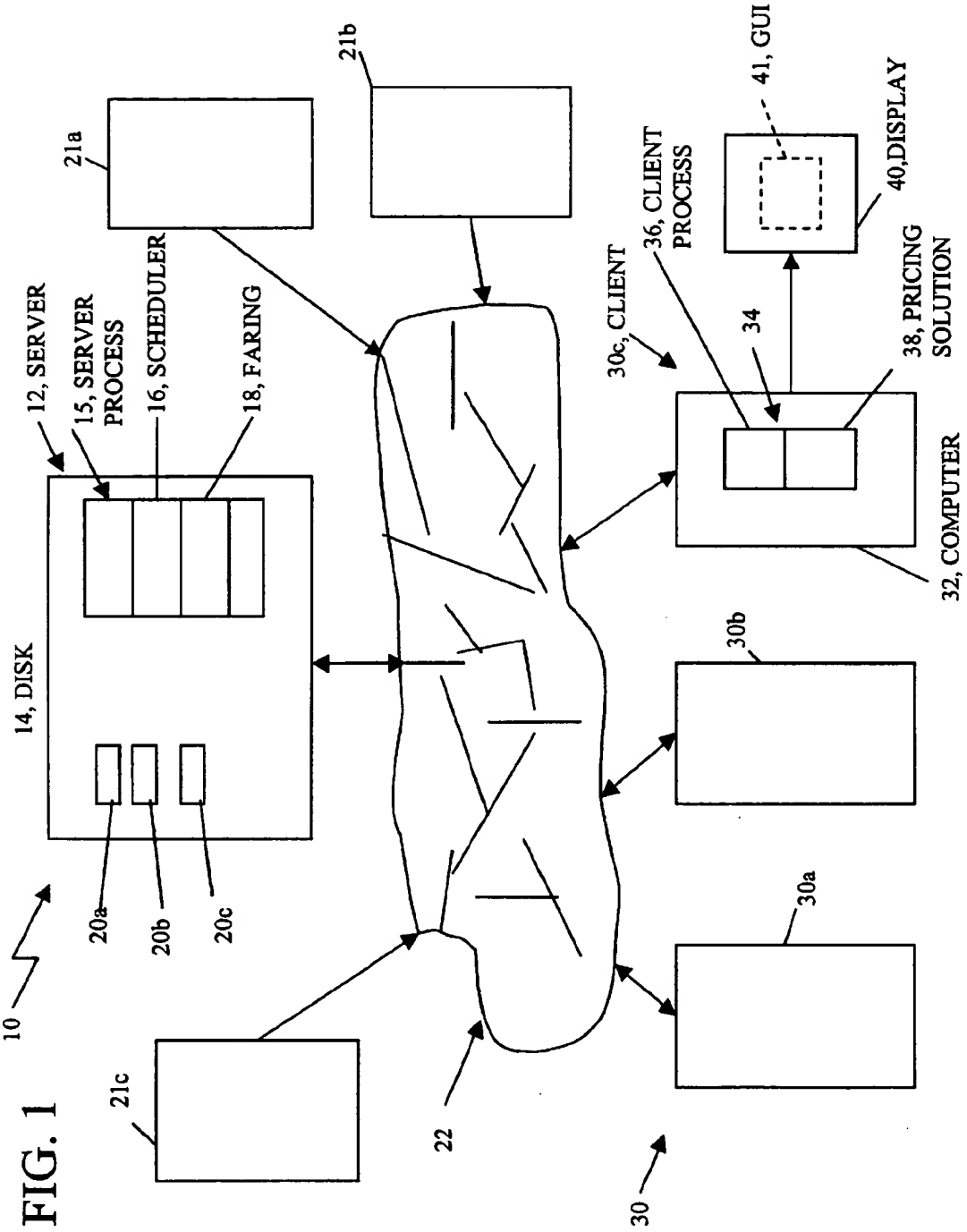

21a
21b
41, GUI
40,DISPLAY
36, CLIENT PROCESS
30c, CLIENT
34
38, PRICING SOLUTION
12, SERVER
15, SERVER PROCESS
16, SCHEDULER
18, FARING
32, COMPUTER
14, DISK
30b
30a
20a
20b
20c
10
21c
22
30

FIG. 1

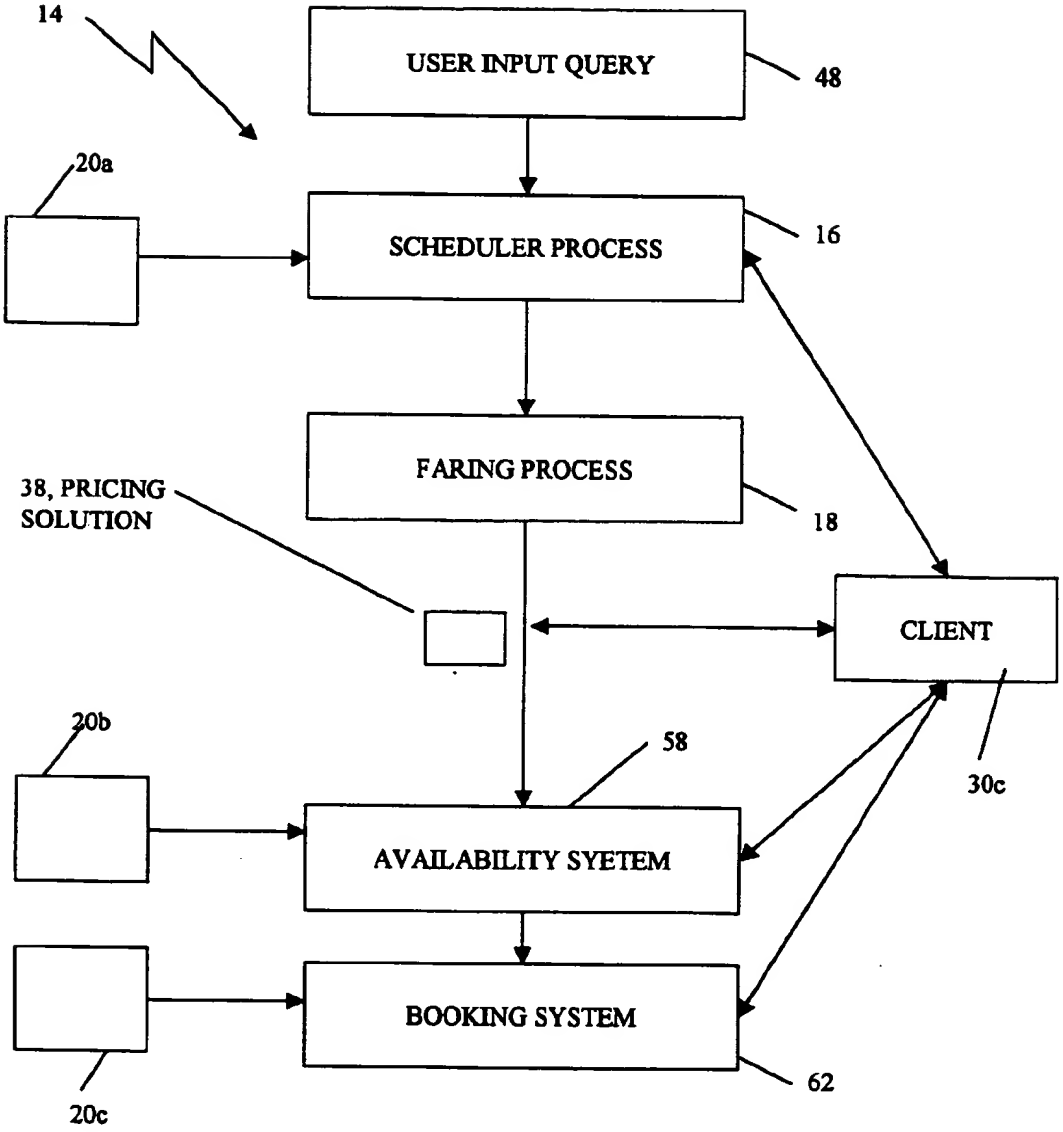

14
USER INPUT QUERY
48
20a
SCHEDULER PROCESS
16
FARING PROCESS
18
38, PRICING SOLUTION
CLIENT
20b
58
30c
AVAILABILITY SYETEM
BOOKING SYSTEM
62
20c

FIG. 2

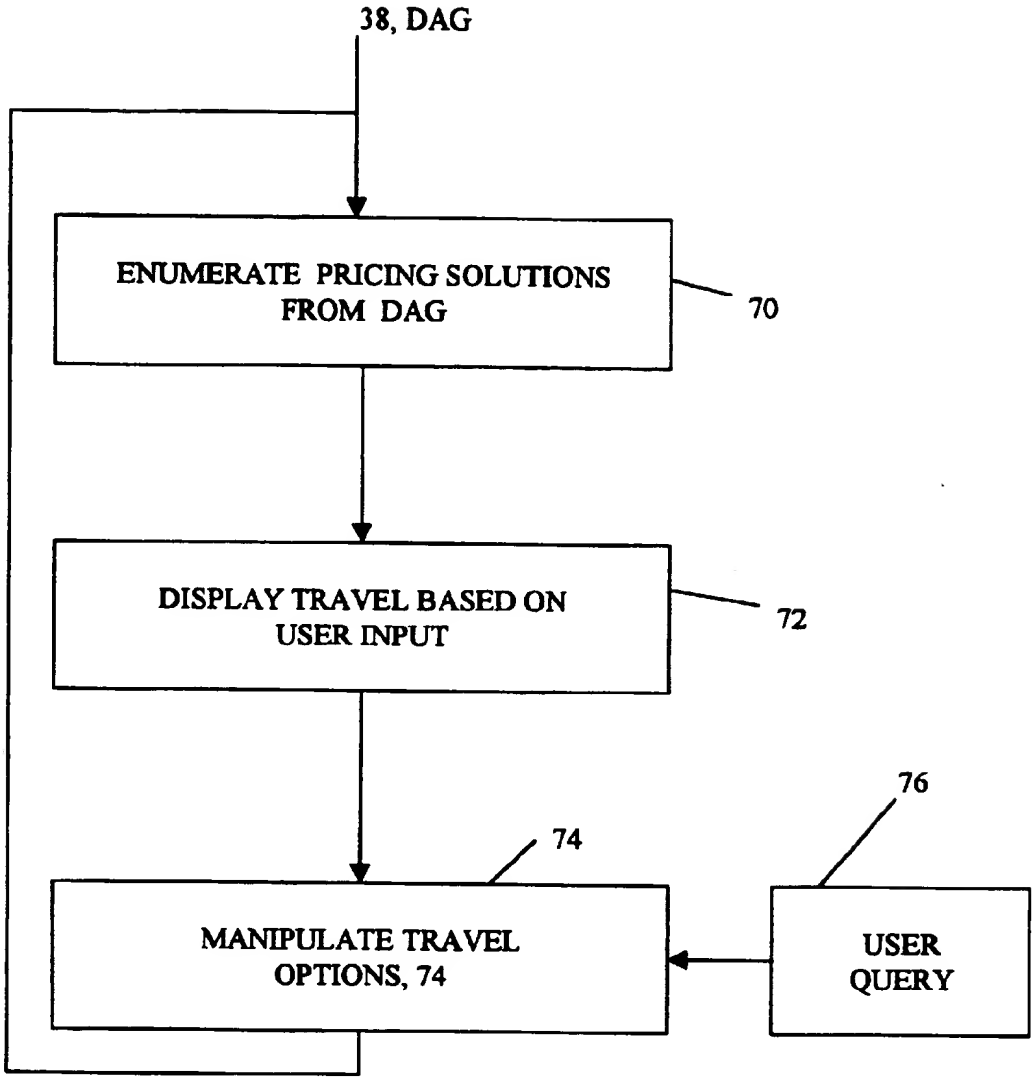

38, DAG
ENUMERATE PRICING SOLUTIONS FROM DAG
70
DISPLAY TRAVEL BASED ON USER INPUT
72
76
74
MANIPULATE TRAVEL OPTIONS, 74
USER QUERY

FIG. 3

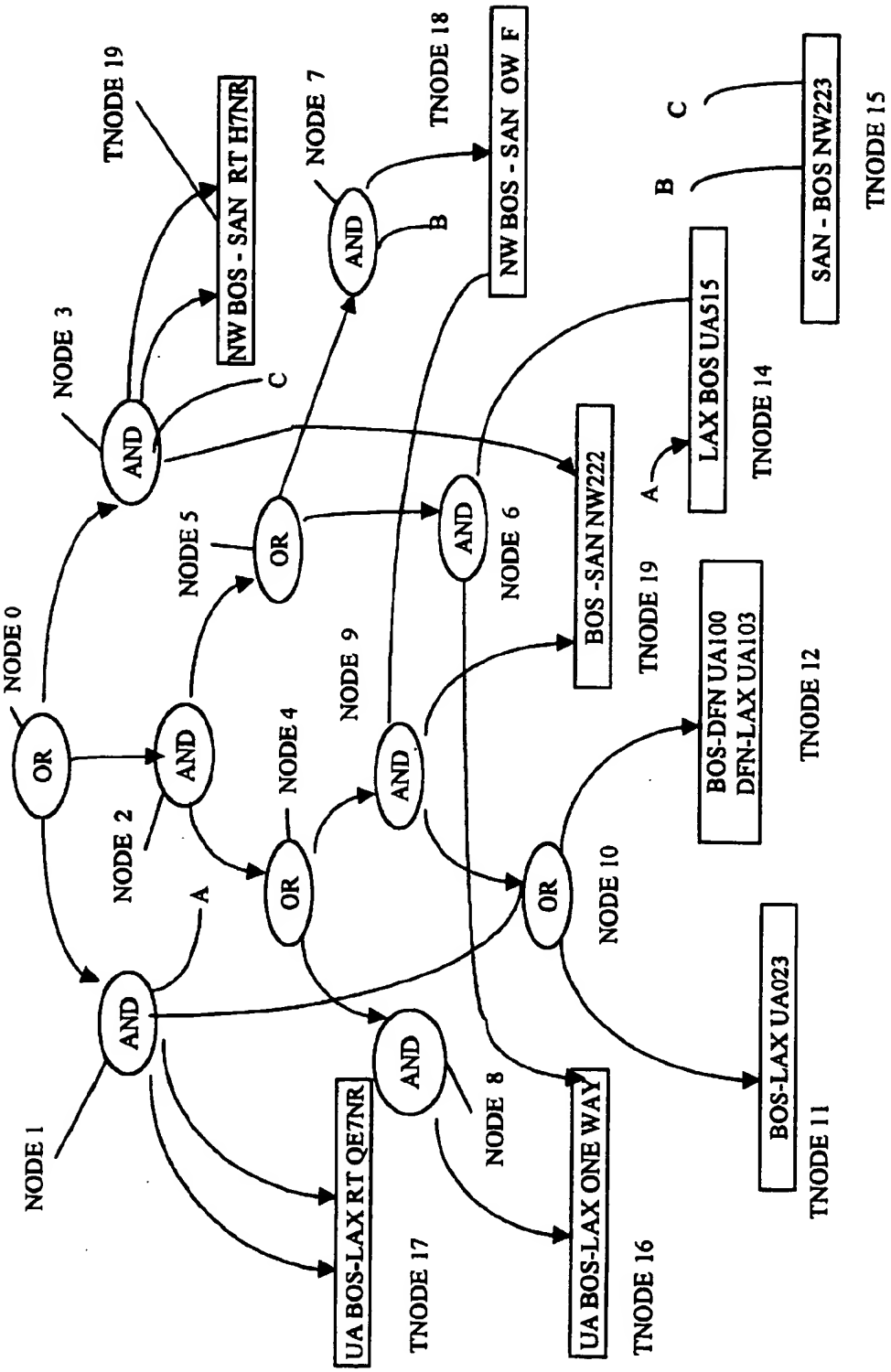
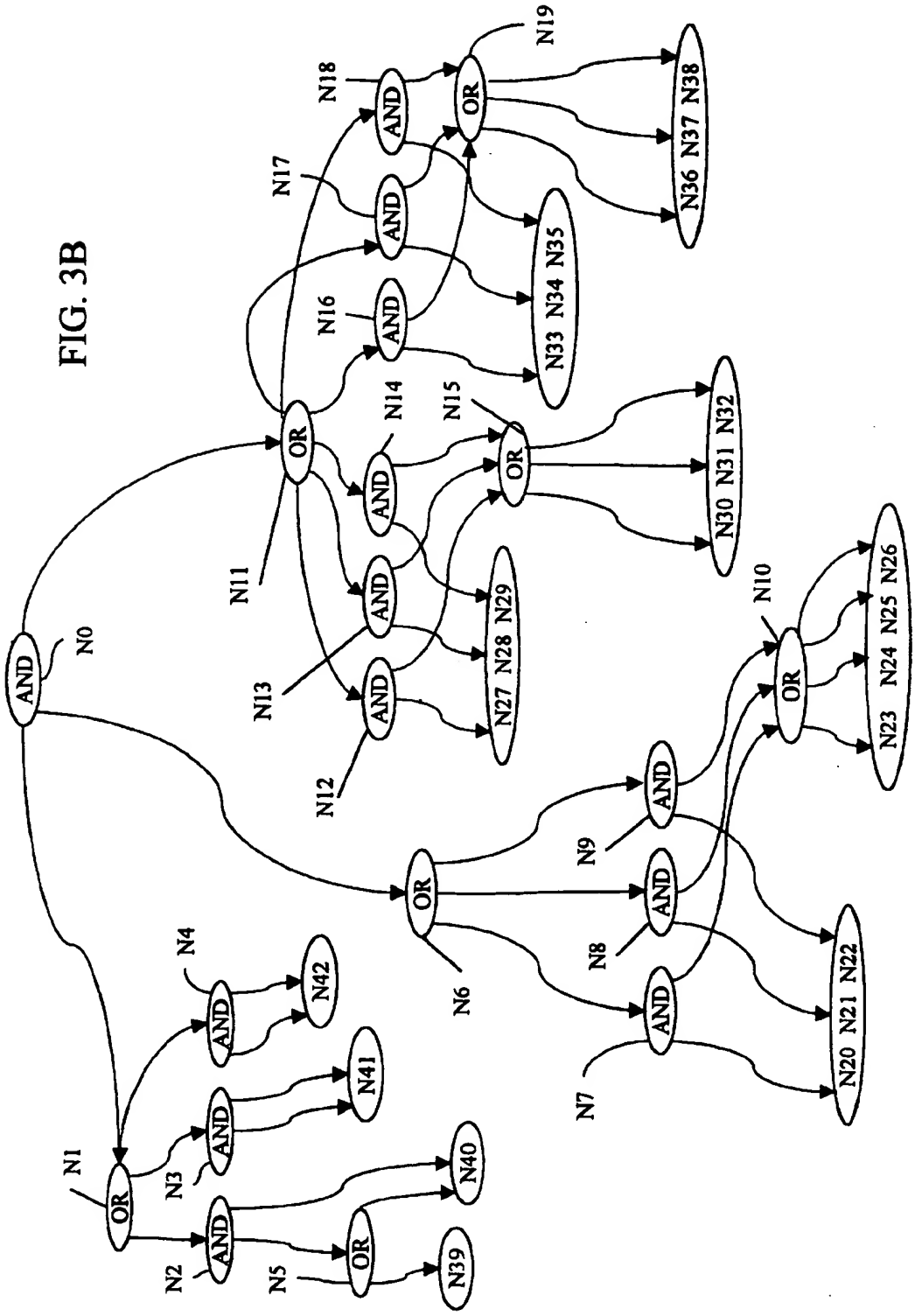

NODE 0
OR
NODE 1
AND
NODE 2
AND
A
NODE 3
AND
C
NODE 4
OR
NODE 5
OR
NODE 6
AND
NODE 7
AND
B
NODE 8
AND
NODE 9
AND
NODE 10
OR
TNODE 19
NW BOS - SAN RT H7NR
TNODE 18
NW BOS - SAN OW F
TNODE 17
UA BOS-LAX RT QE7NR
TNODE 16
UA BOS-LAX ONE WAY
BOS -SAN NW222
TNODE 19
A
LAX BOS UA515
TNODE 14
B
C
SAN - BOS NW223
TNODE 15
BOS-DFN UA100
DFN-LAX UA103
TNODE 12
BOS-LAX UA023
TNODE 11

FIG. 3A

FIG. 3B


AND
N0
OR
N1
AND
N2
AND
N3
AND
N4
N5
OR
N6
OR
N7
AND
N8
AND
N9
OR
N10
N11
AND
N12
AND
N13
AND
N14
OR
N15
AND
N16
AND
N17
AND
N18
OR
N19
N20 N21 N22
N23 N24 N25 N26
N27 N28 N29
N30 N31 N32
N33 N34 N35
N36 N37 N38
N39
N40
N41
N42

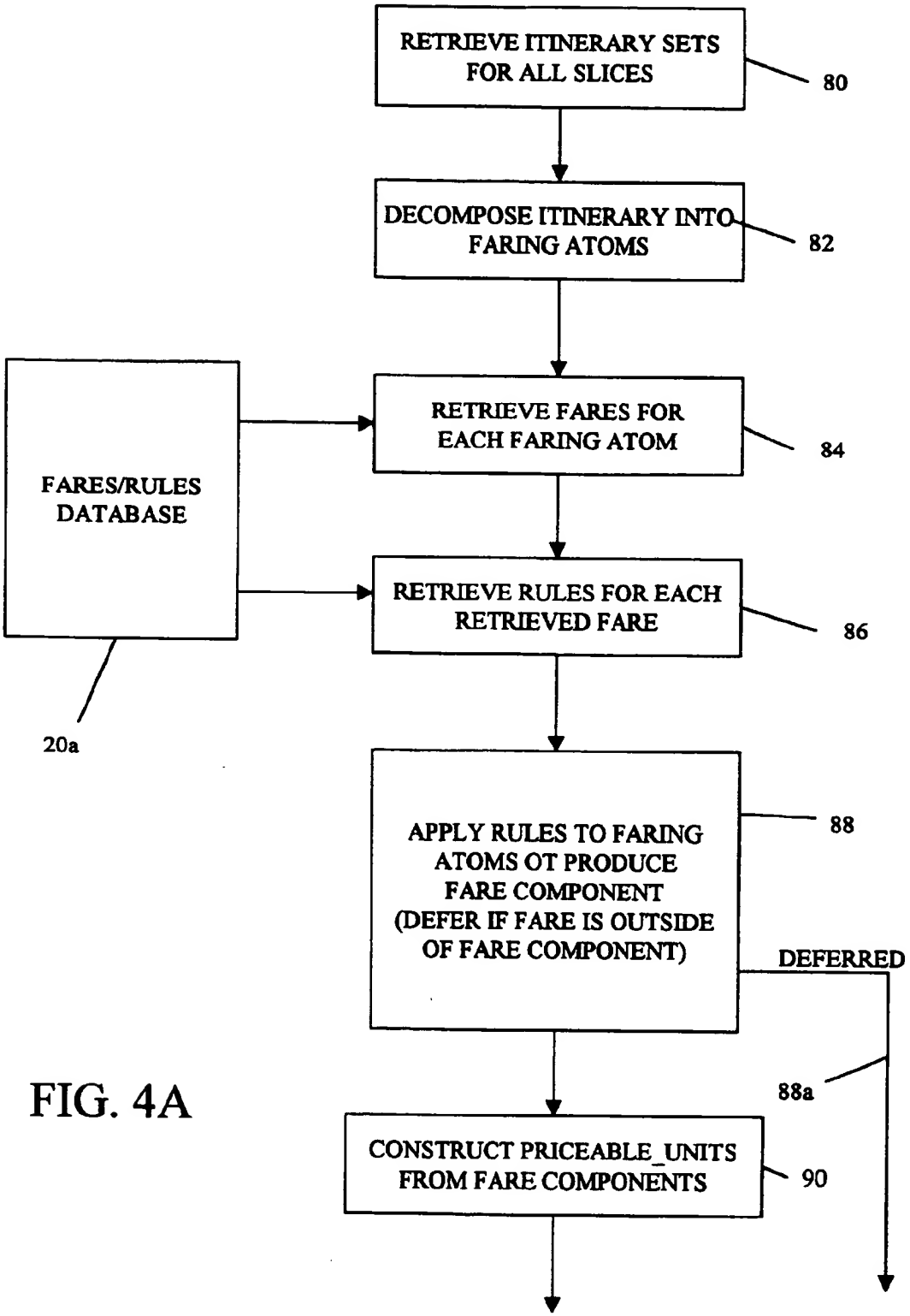

RETRIEVE ITINERARY SETS FOR ALL SLICES
80
DECOMPOSE ITINERARY INTO FARING ATOMS
82
FARES/RULES DATABASE
RETRIEVE FARES FOR EACH FARING ATOM
84
RETRIEVE RULES FOR EACH RETRIEVED FARE
86
20a
APPLY RULES TO FARING ATOMS OT PRODUCE FARE COMPONENT (DEFER IF FARE IS OUTSIDE OF FARE COMPONENT)
88
DEFERRED
88a
CONSTRUCT PRICEABLE_UNITS FROM FARE COMPONENTS
90

FIG. 4A

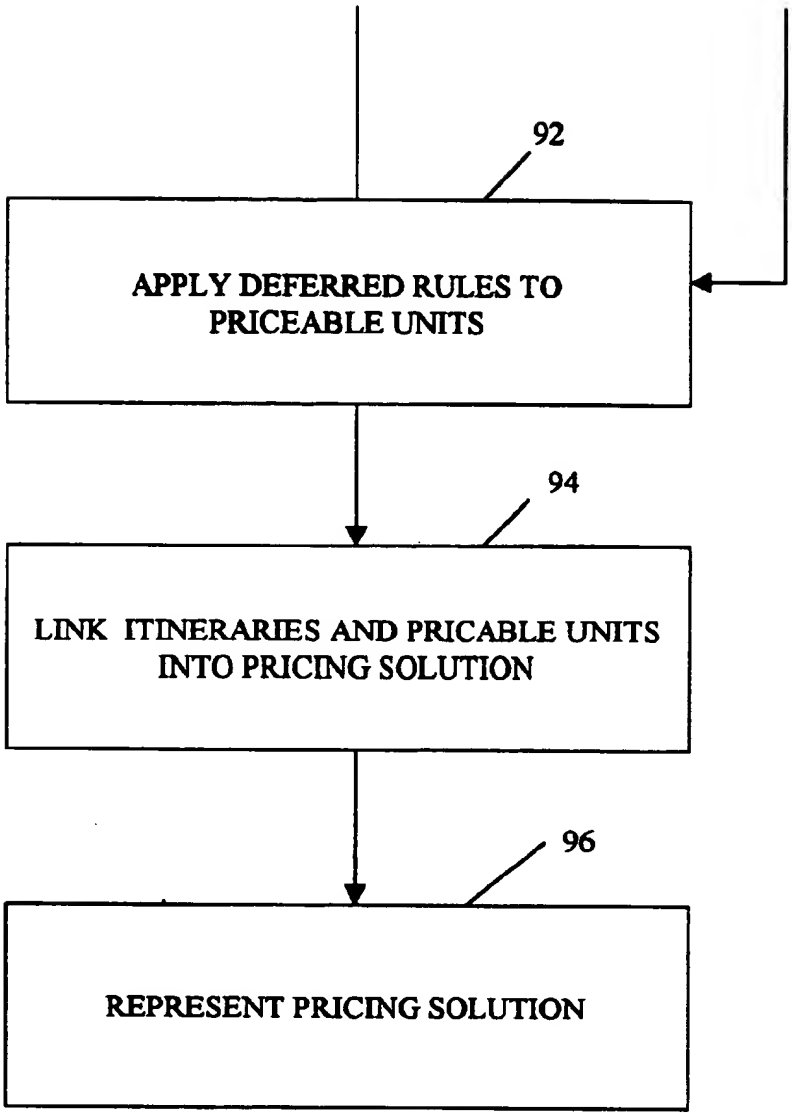

92
APPLY DEFERRED RULES TO PRICEABLE UNITS
94
LINK ITINERARIES AND PRICABLE UNITS INTO PRICING SOLUTION
96
REPRESENT PRICING SOLUTION

FIG. 4B

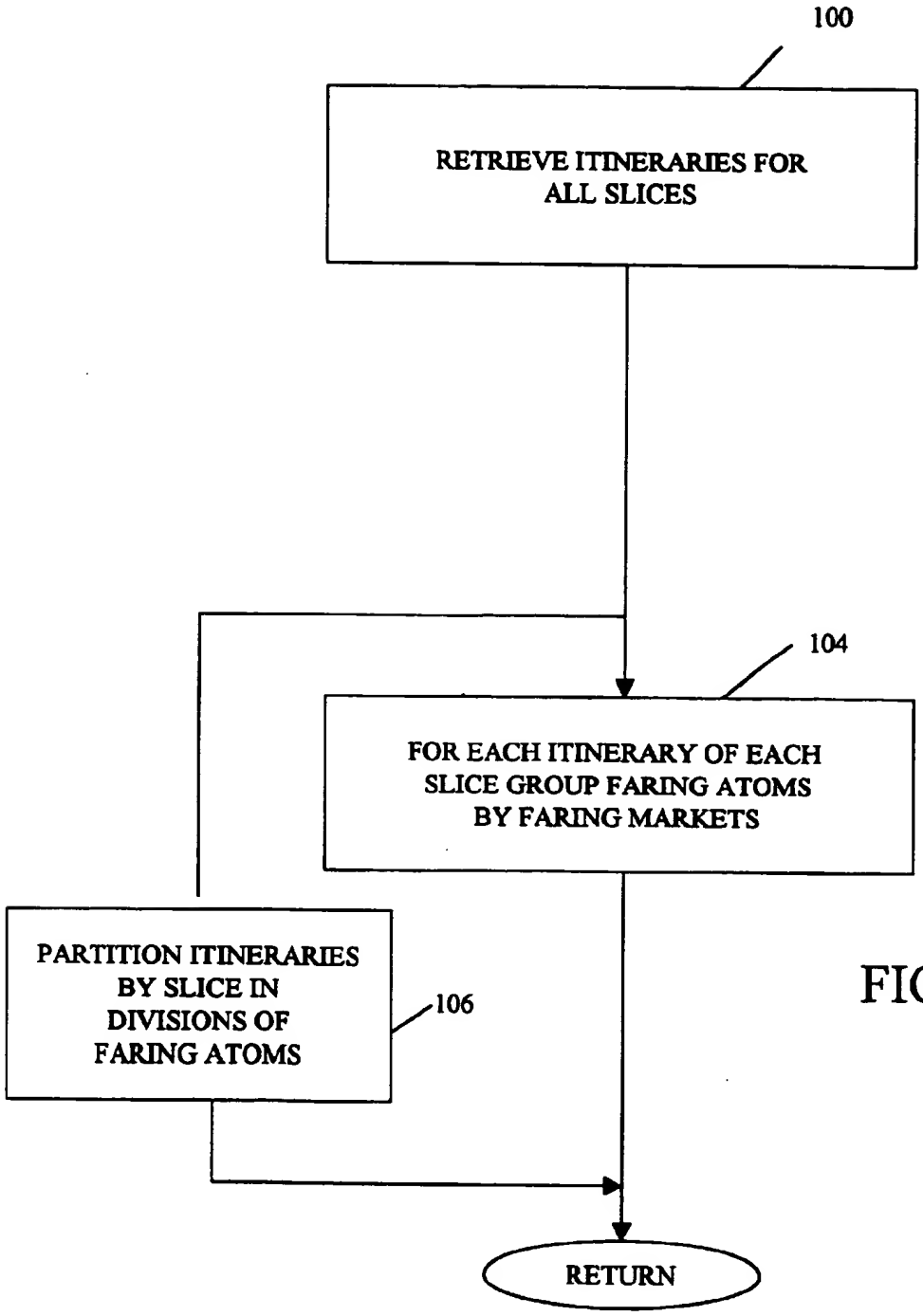

100
RETRIEVE ITINERARIES FOR ALL SLICES
104
FOR EACH ITINERARY OF EACH SLICE GROUP FARING ATOMS BY FARING MARKETS
PARTITION ITINERARIES BY SLICE IN DIVISIONS OF FARING ATOMS
106
RETURN

FIG. 5

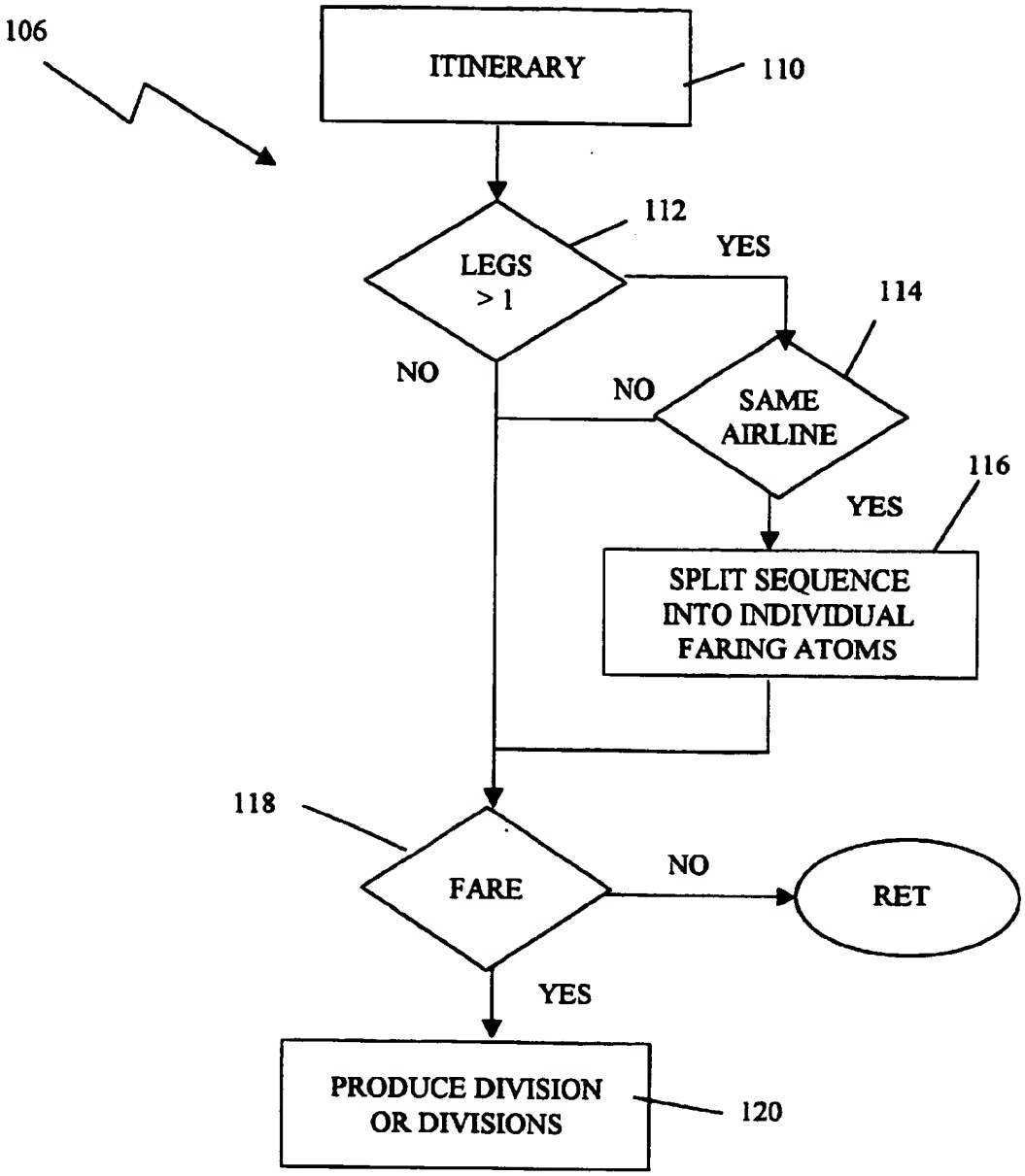

106
ITINERARY
110
112
LEGS
> 1
YES
114
NO
NO
SAME
AIRLINE
116
YES
SPLIT SEQUENCE
INTO INDIVIDUAL
FARING ATOMS
118
FARE
NO
RET
YES
PRODUCE DIVISION
OR DIVISIONS
120

FIG. 6

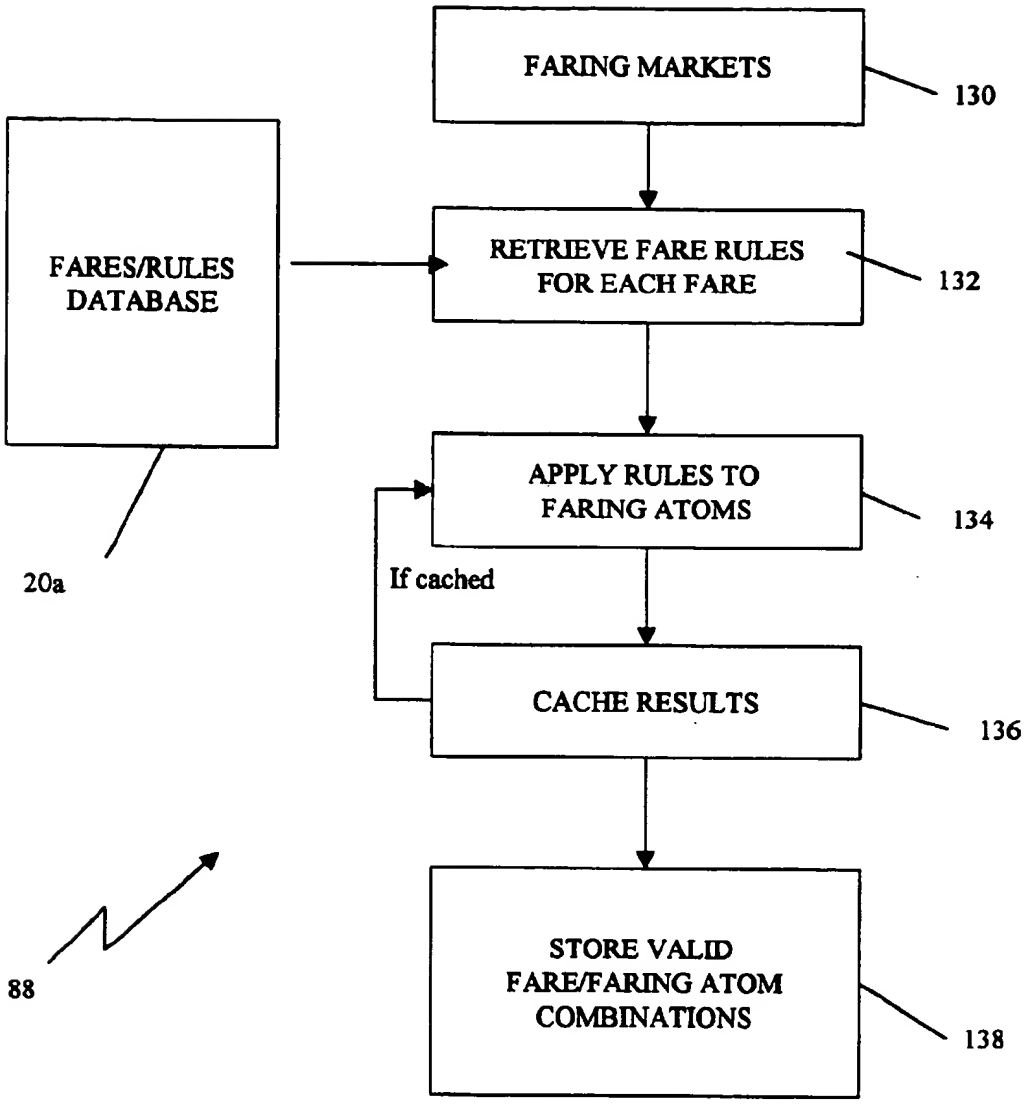

FARING MARKETS
130
FARES/RULES DATABASE
RETRIEVE FARE RULES FOR EACH FARE
132
APPLY RULES TO FARING ATOMS
134
If cached
CACHE RESULTS
136
20a
88
STORE VALID FARE/FARING ATOM COMBINATIONS
138

FIG. 7

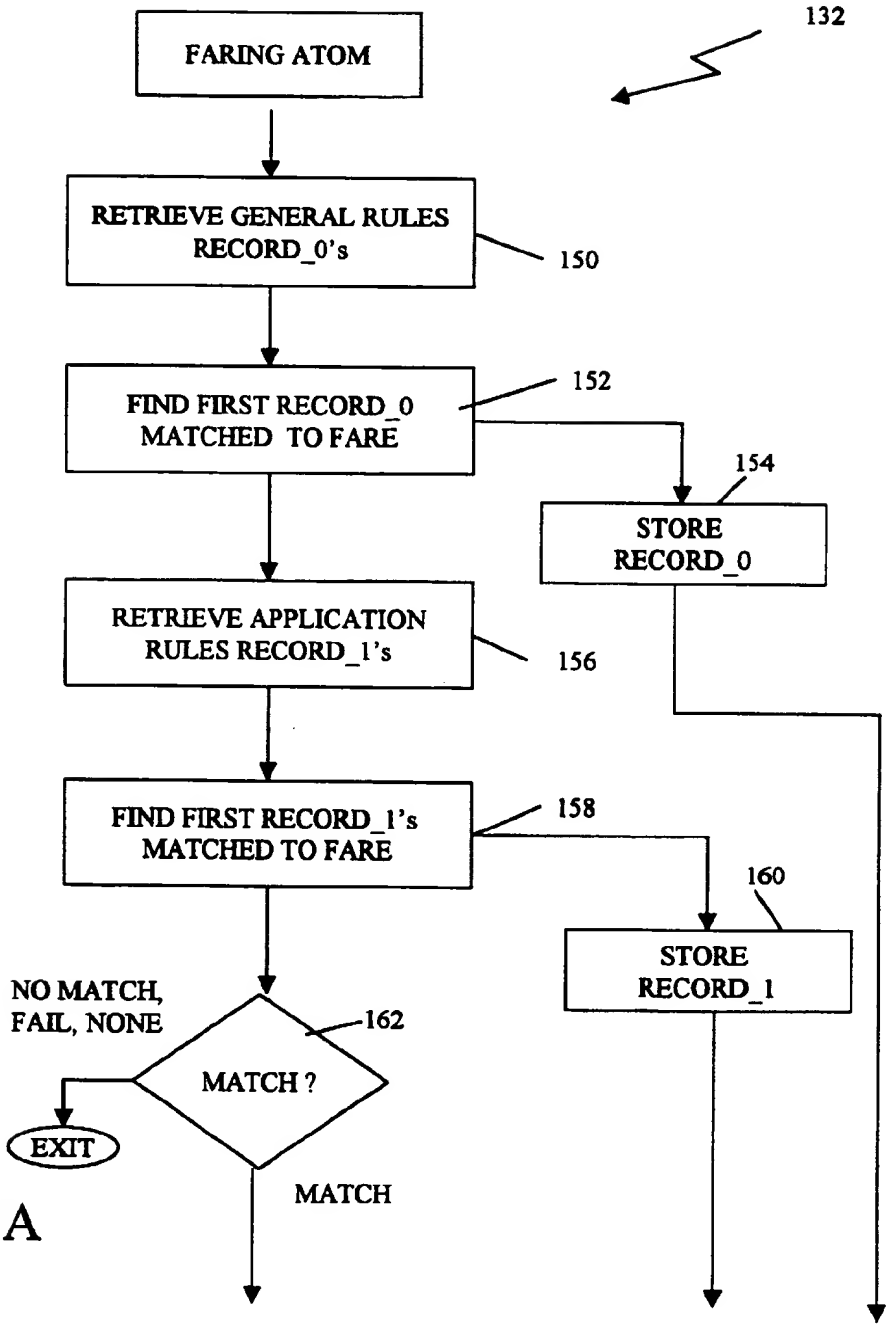

FARING ATOM
132
RETRIEVE GENERAL RULES RECORD_0's
150
FIND FIRST RECORD_0 MATCHED TO FARE
152
154
STORE RECORD_0
RETRIEVE APPLICATION RULES RECORD_1's
156
FIND FIRST RECORD_1's MATCHED TO FARE
158
160
STORE RECORD_1
NO MATCH, FAIL, NONE
162
MATCH ?
EXIT
MATCH

FIG. 8A

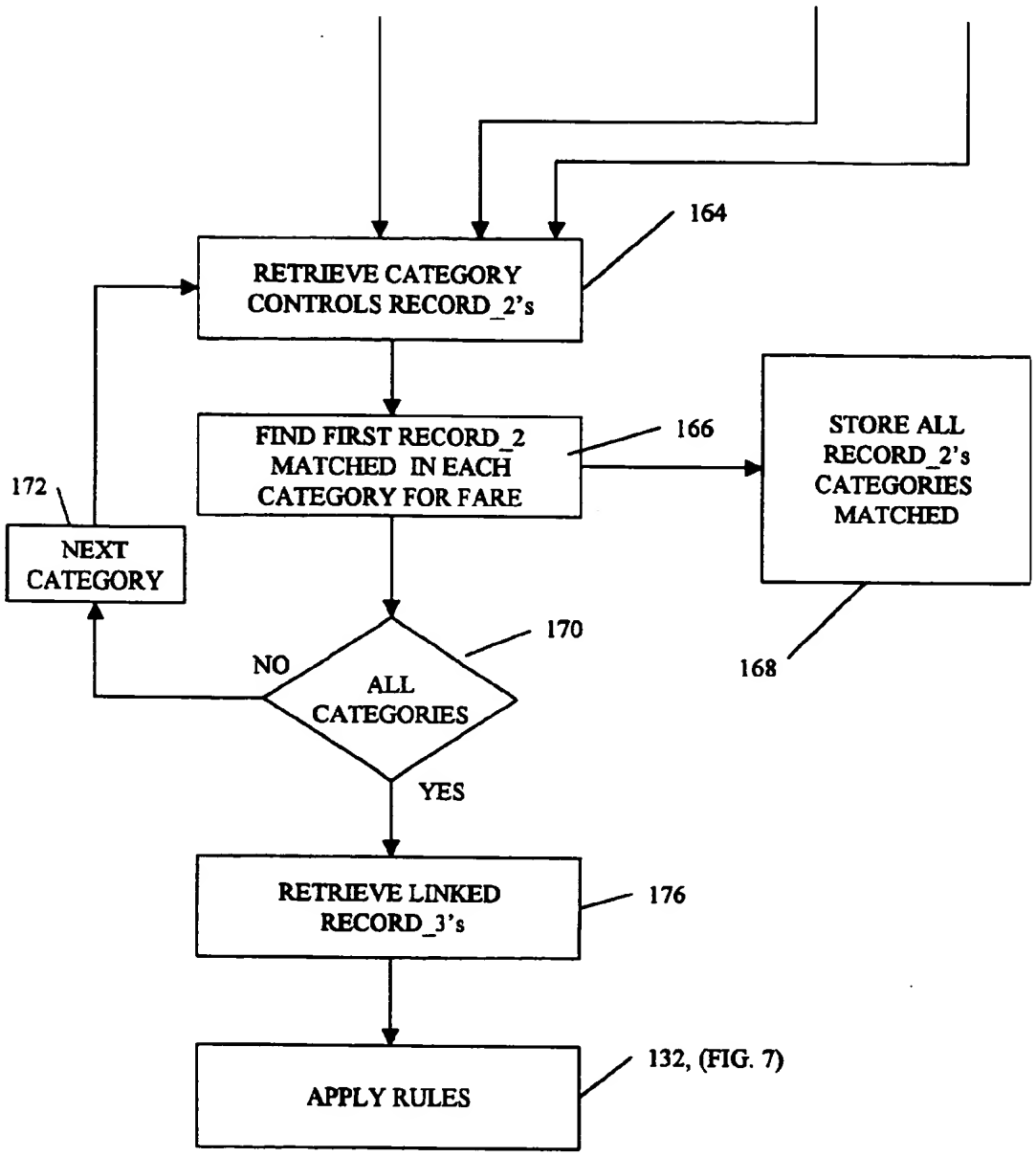

RETRIEVE CATEGORY CONTROLS RECORD_2's
164
FIND FIRST RECORD_2 MATCHED IN EACH CATEGORY FOR FARE
166
STORE ALL RECORD_2's CATEGORIES MATCHED
168
172
NEXT CATEGORY
NO
ALL CATEGORIES
170
YES
RETRIEVE LINKED RECORD_3's
176
APPLY RULES
132, (FIG. 7)

FIG. 8B

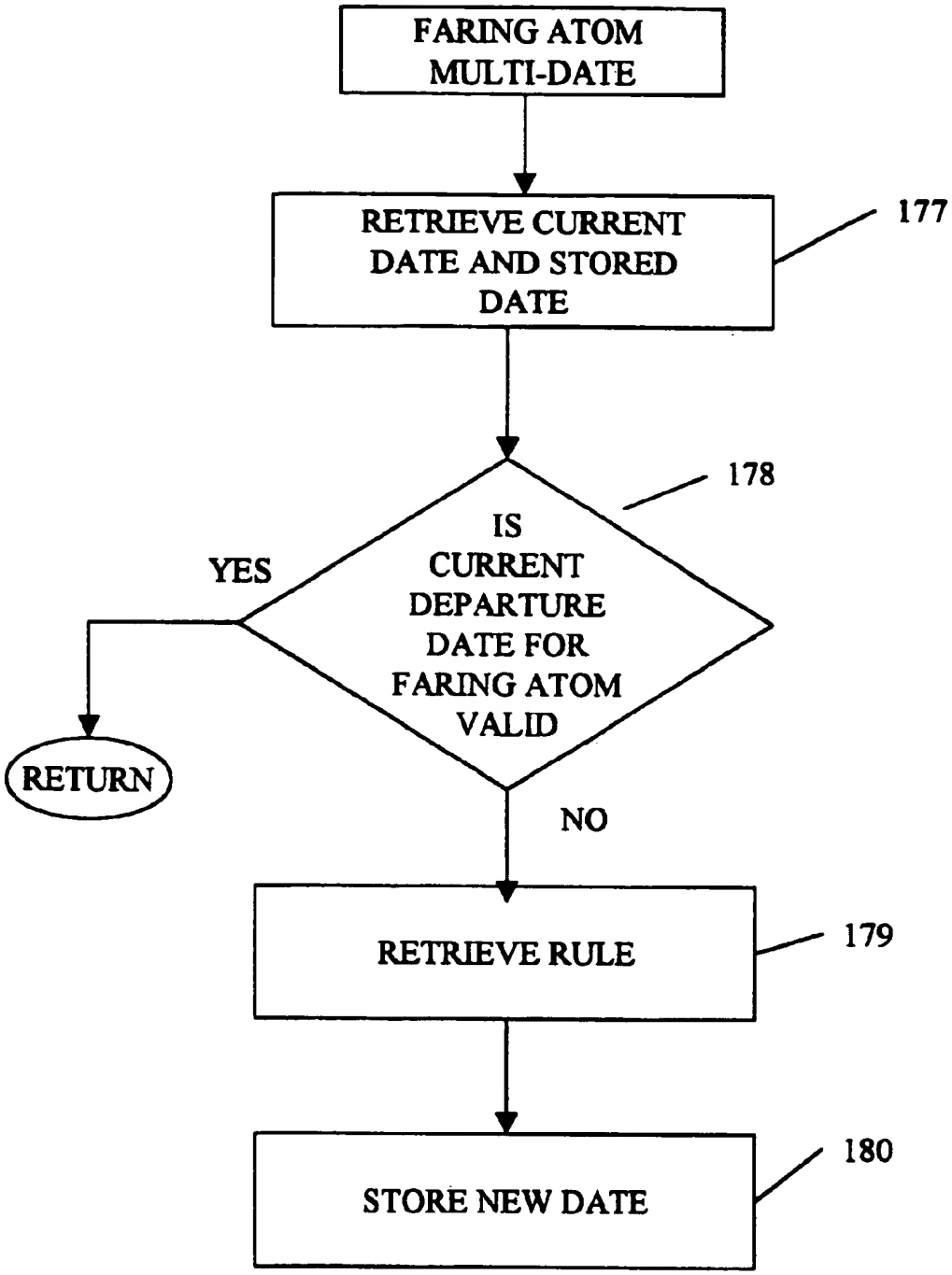

FARING ATOM MULTI-DATE
RETRIEVE CURRENT DATE AND STORED DATE
177
178
IS CURRENT DEPARTURE DATE FOR FARING ATOM VALID
YES
RETURN
NO
RETRIEVE RULE
179
STORE NEW DATE
180

FIG. 8C

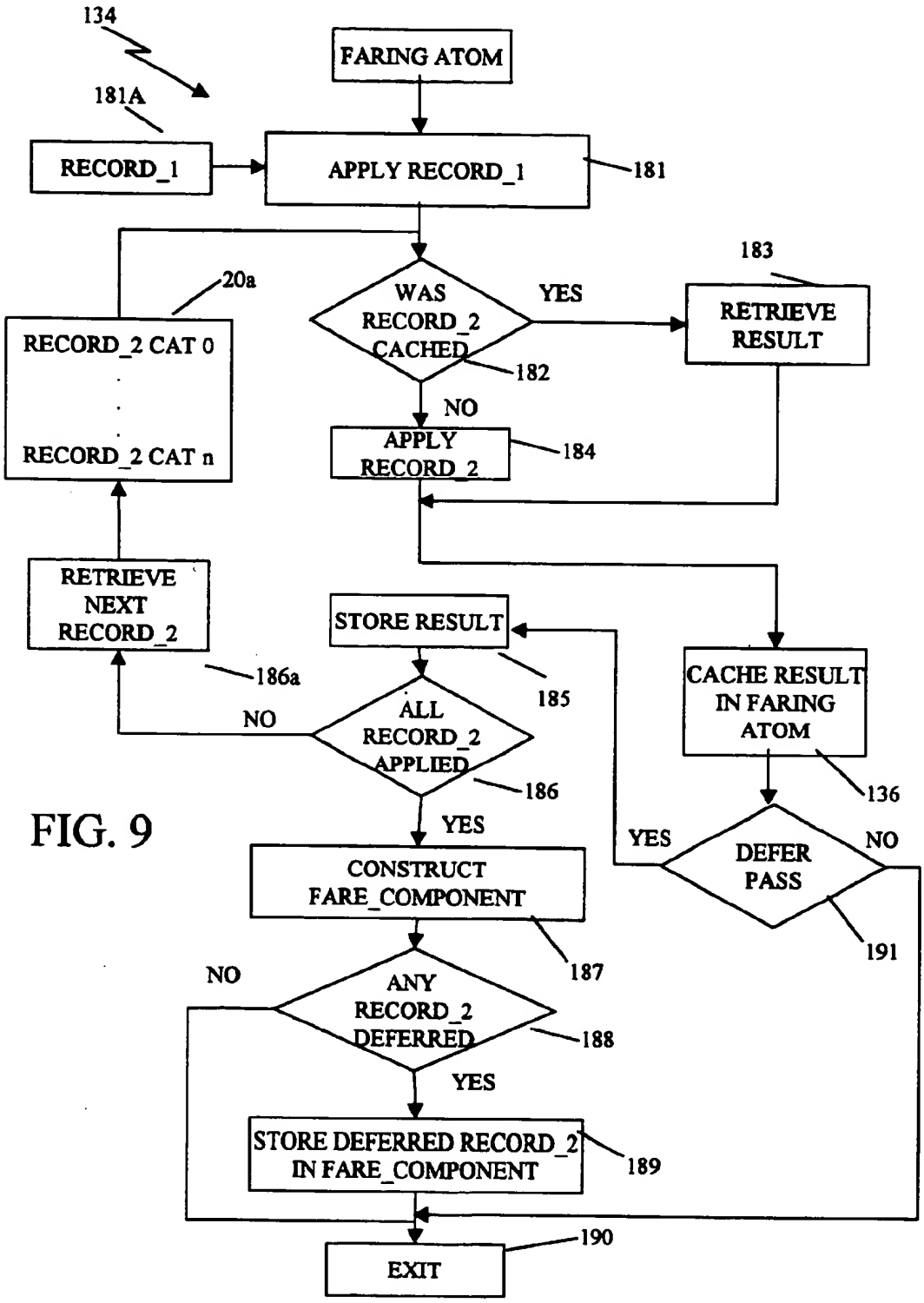

134
FARING ATOM
181A
RECORD_1
APPLY RECORD_1
181
183
20a
WAS RECORD_2 CACHED
YES
RETRIEVE RESULT
RECORD_2 CAT 0
RECORD_2 CAT n
182
NO
APPLY RECORD_2
184
RETRIEVE NEXT RECORD_2
STORE RESULT
CACHE RESULT IN FARING ATOM
186a
185
NO
ALL RECORD_2 APPLIED
186
136
YES
YES
DEFER PASS
NO
CONSTRUCT FARE_COMPONENT
191
187
NO
ANY RECORD_2 DEFERRED
188
YES
STORE DEFERRED RECORD_2 IN FARE_COMPONENT
189
190
EXIT

FIG. 9

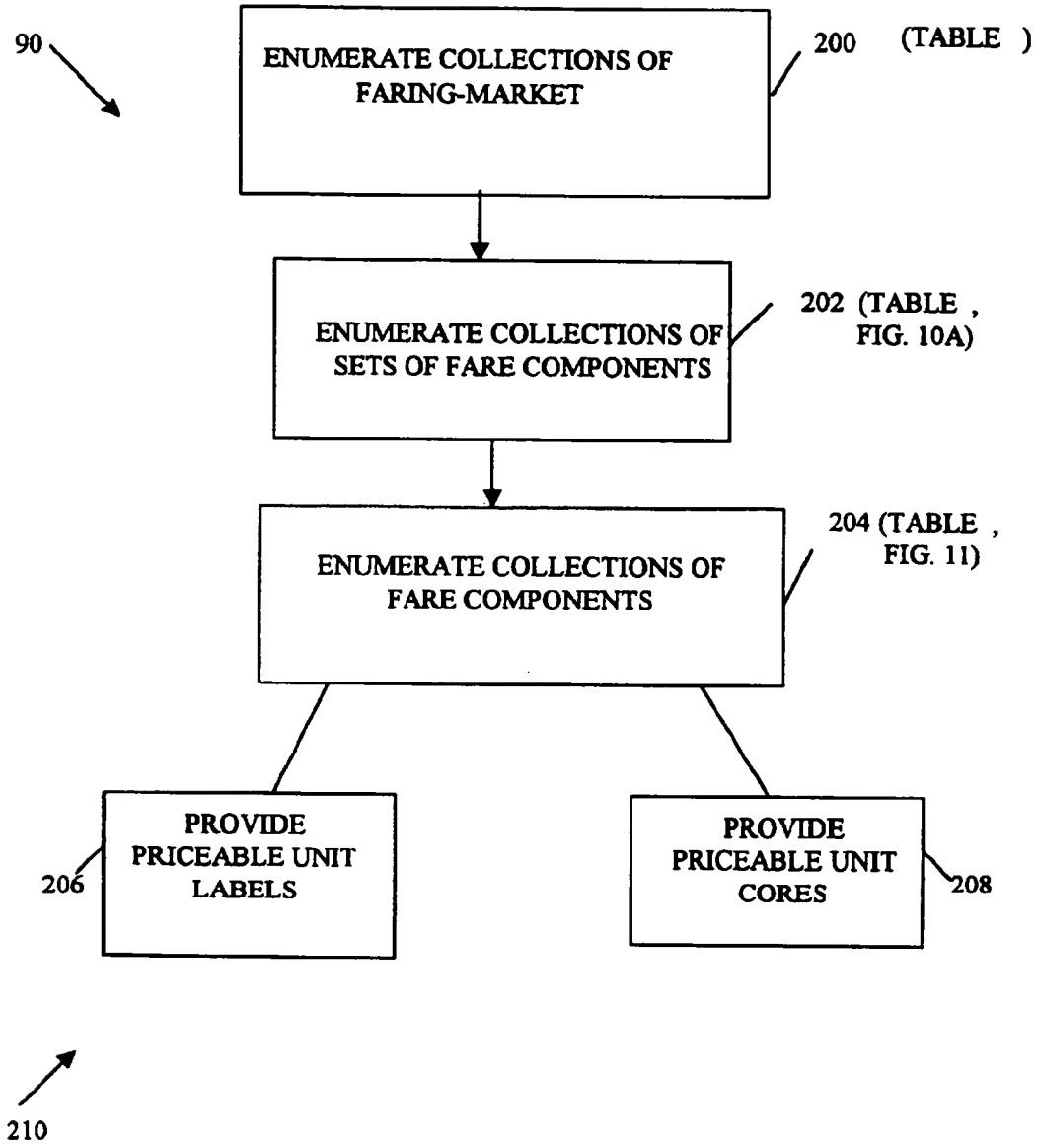

90
ENUMERATE COLLECTIONS OF FARING-MARKET
200 (TABLE)
ENUMERATE COLLECTIONS OF SETS OF FARE COMPONENTS
202 (TABLE , FIG. 10A)
ENUMERATE COLLECTIONS OF FARE COMPONENTS
204 (TABLE , FIG. 11)
PROVIDE PRICEABLE UNIT LABELS
206
PROVIDE PRICEABLE UNIT CORES
208
210

FIG. 10

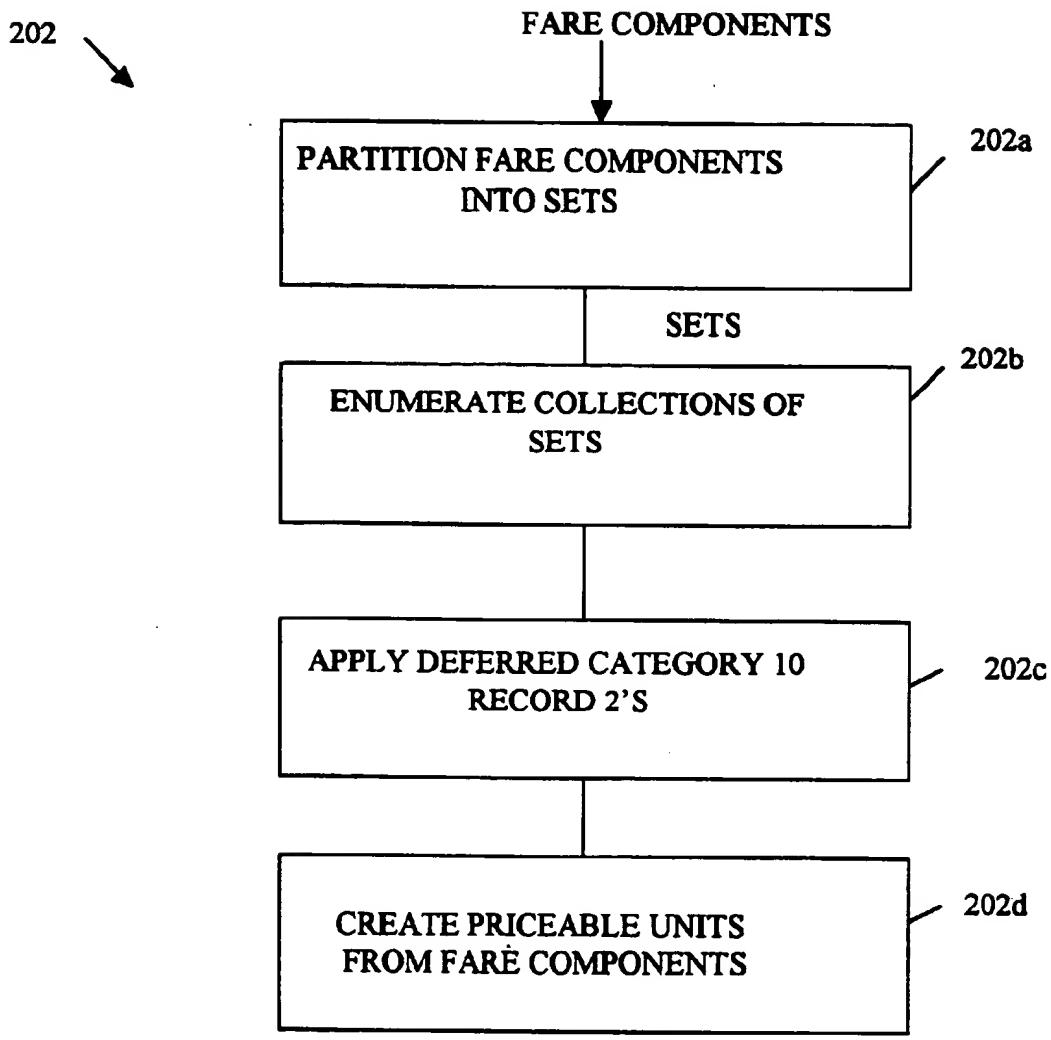

202
FARE COMPONENTS
PARTITION FARE COMPONENTS INTO SETS
202a
SETS
ENUMERATE COLLECTIONS OF SETS
202b
APPLY DEFERRED CATEGORY 10 RECORD 2'S
202c
CREATE PRICEABLE UNITS FROM FARE COMPONENTS
202d

FIG. 10A

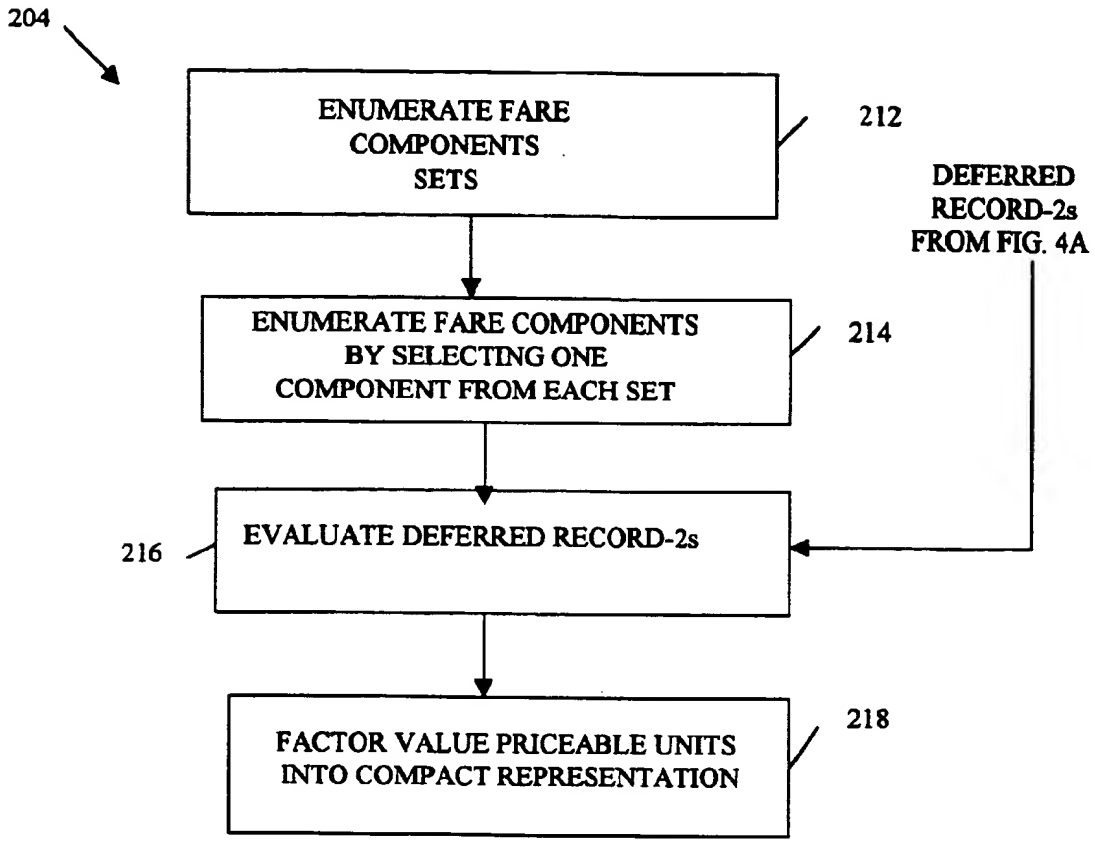

204
ENUMERATE FARE COMPONENTS SETS
212
DEFERRED RECORD-2s FROM FIG. 4A
ENUMERATE FARE COMPONENTS BY SELECTING ONE COMPONENT FROM EACH SET
214
EVALUATE DEFERRED RECORD-2s
216
FACTOR VALUE PRICEABLE UNITS INTO COMPACT REPRESENTATION
218

FIG. 11

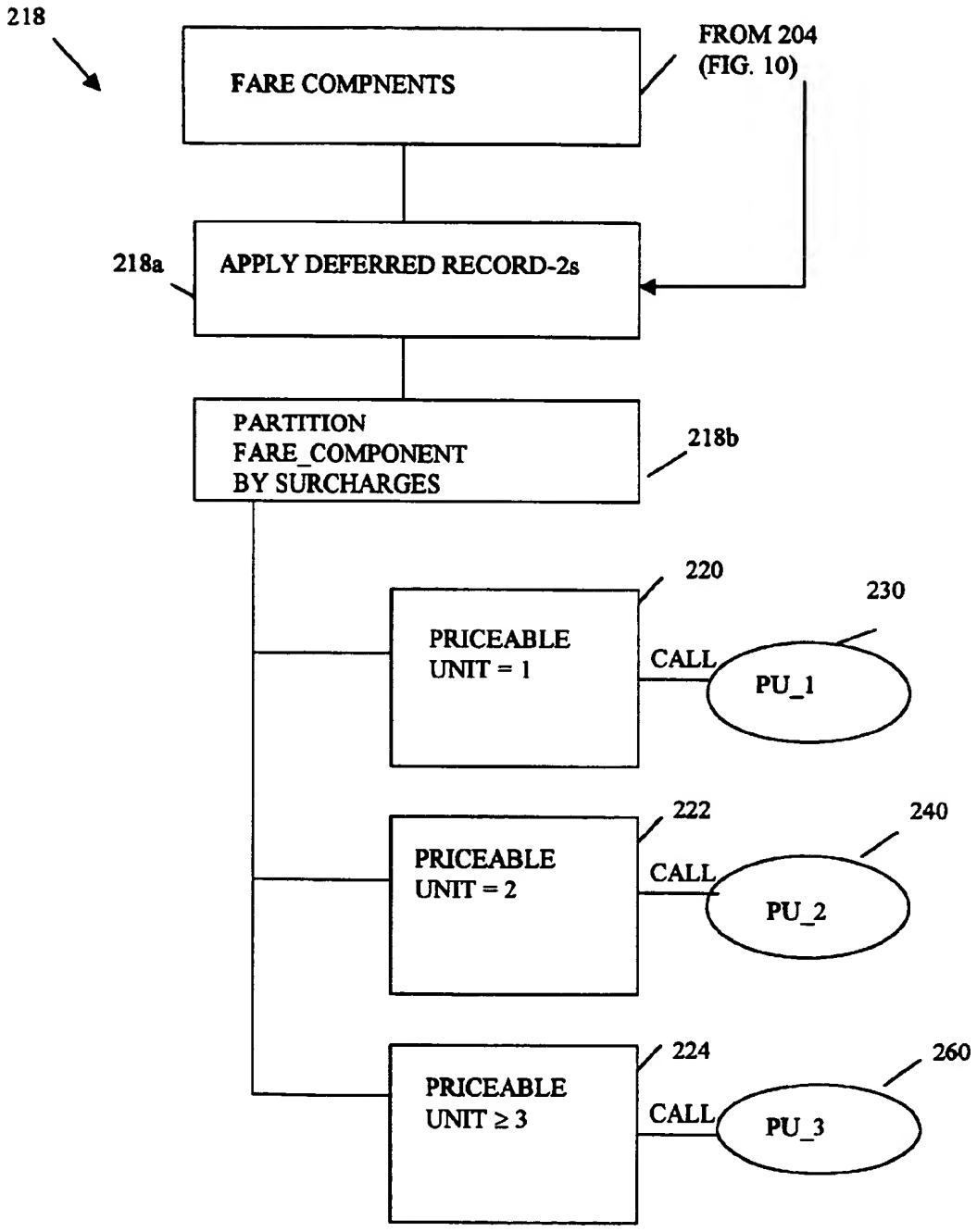

218
FARE COMPNENTS
FROM 204
(FIG. 10)
218a
APPLY DEFERRED RECORD-2s
PARTITION
FARE_COMPONENT
BY SURCHARGES
218b
220
230
PRICEABLE
UNIT = 1
CALL
PU_1
222
240
PRICEABLE
UNIT = 2
CALL
PU_2
224
260
PRICEABLE
UNIT ≥ 3
CALL
PU_3

FIG. 12

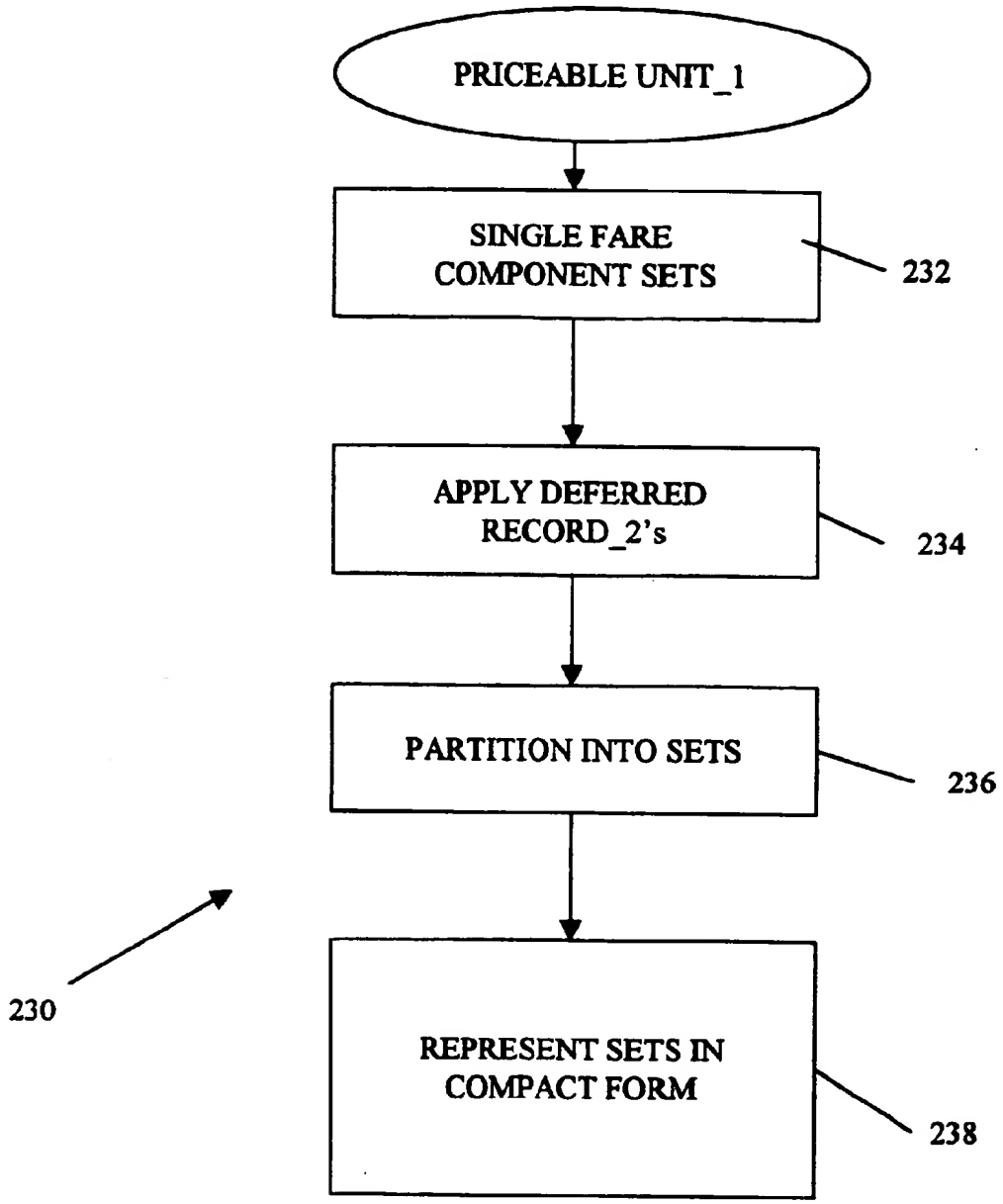

PRICEABLE UNIT_1
SINGLE FARE COMPONENT SETS
232
APPLY DEFERRED RECORD_2's
234
PARTITION INTO SETS
236
230
REPRESENT SETS IN COMPACT FORM
238

FIG. 13

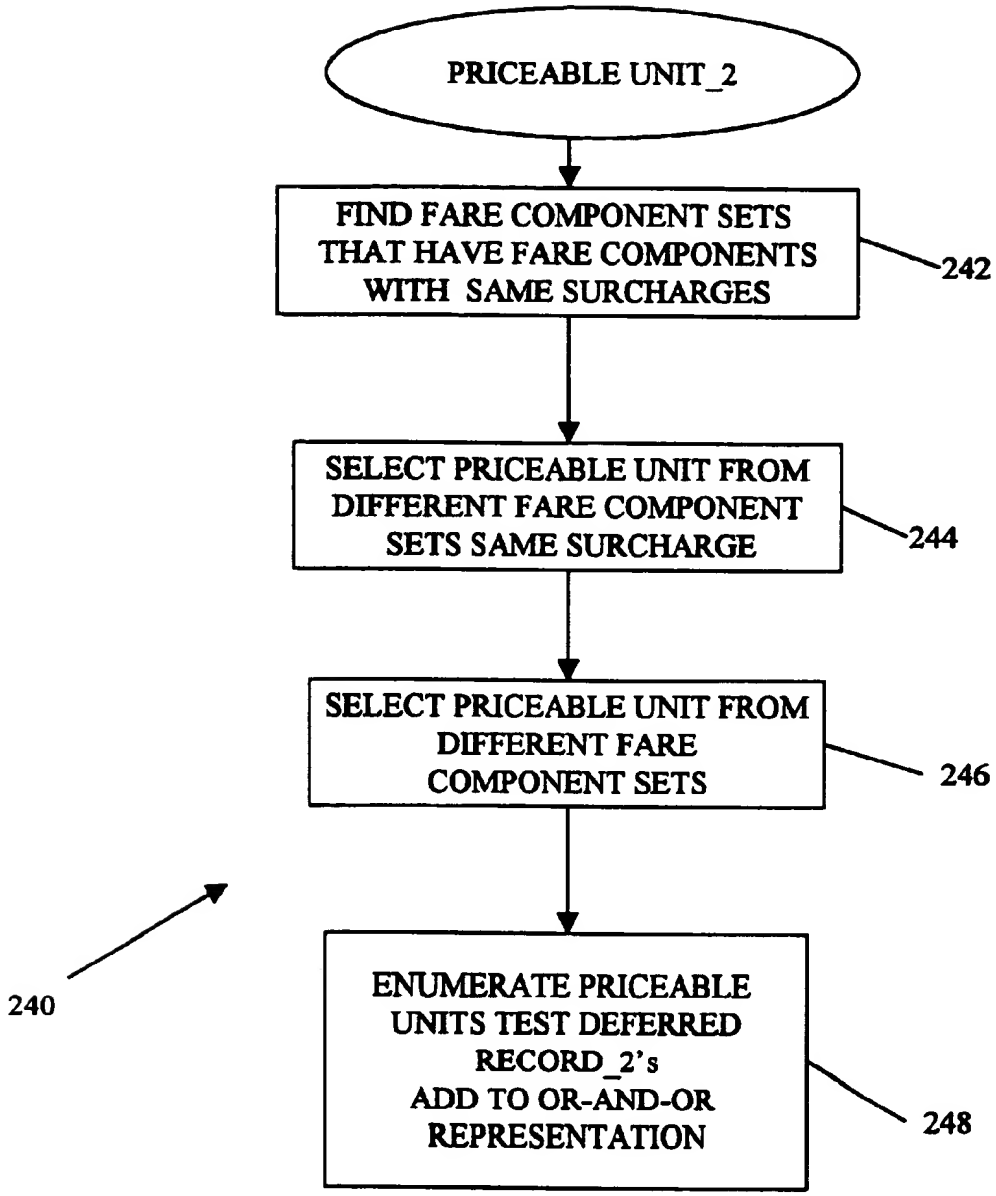

PRICEABLE UNIT_2
FIND FARE COMPONENT SETS THAT HAVE FARE COMPONENTS WITH SAME SURCHARGES
242
SELECT PRICEABLE UNIT FROM DIFFERENT FARE COMPONENT SETS SAME SURCHARGE
244
SELECT PRICEABLE UNIT FROM DIFFERENT FARE COMPONENT SETS
246
240
ENUMERATE PRICEABLE UNITS TEST DEFERRED RECORD_2's ADD TO OR-AND-OR REPRESENTATION
248

FIG. 14

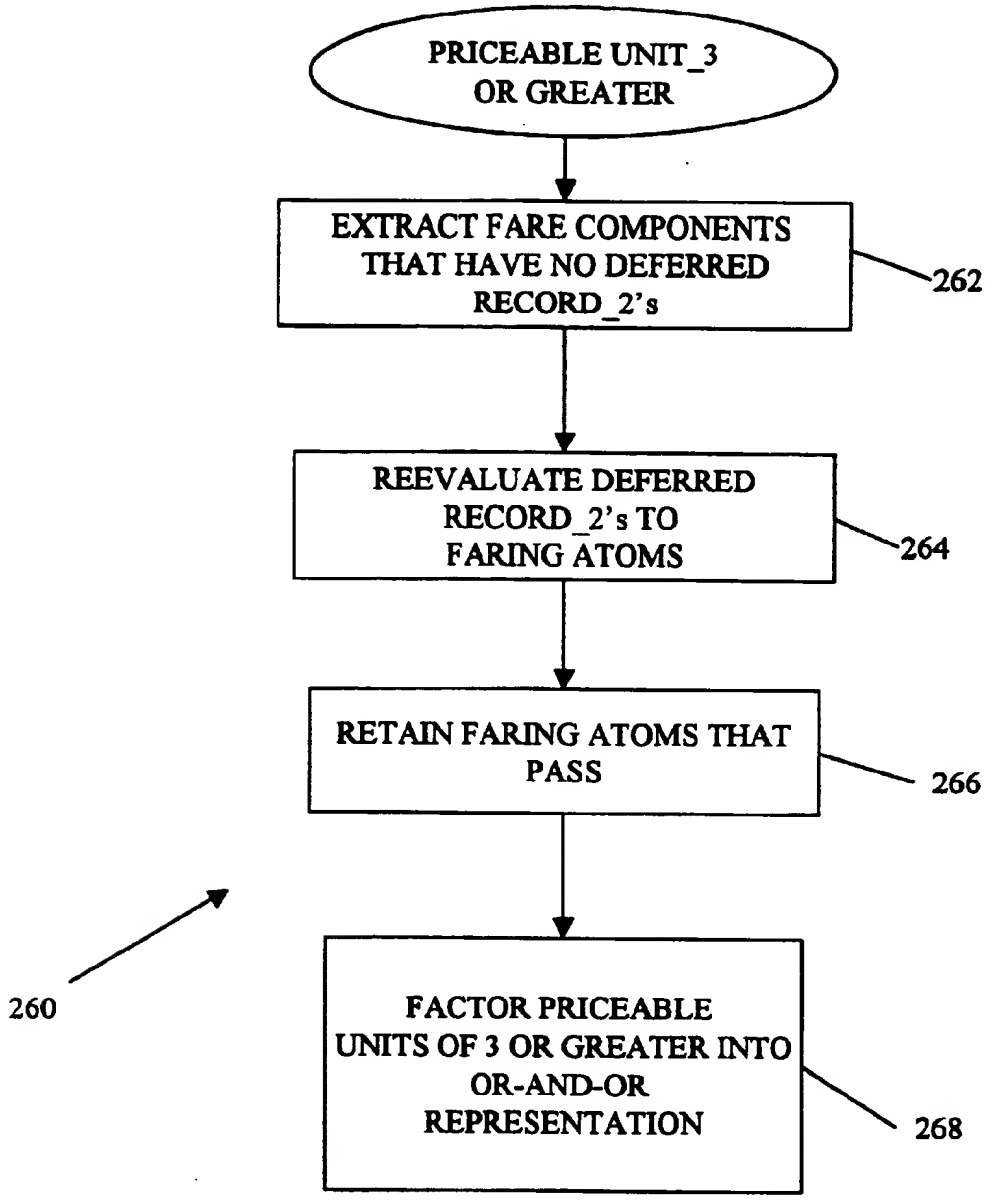

PRICEABLE UNIT_3 OR GREATER
EXTRACT FARE COMPONENTS THAT HAVE NO DEFERRED RECORD_2's
262
REEVALUATE DEFERRED RECORD_2's TO FARING ATOMS
264
RETAIN FARING ATOMS THAT PASS
266
260
FACTOR PRICEABLE UNITS OF 3 OR GREATER INTO OR-AND-OR REPRESENTATION
268

FIG. 15

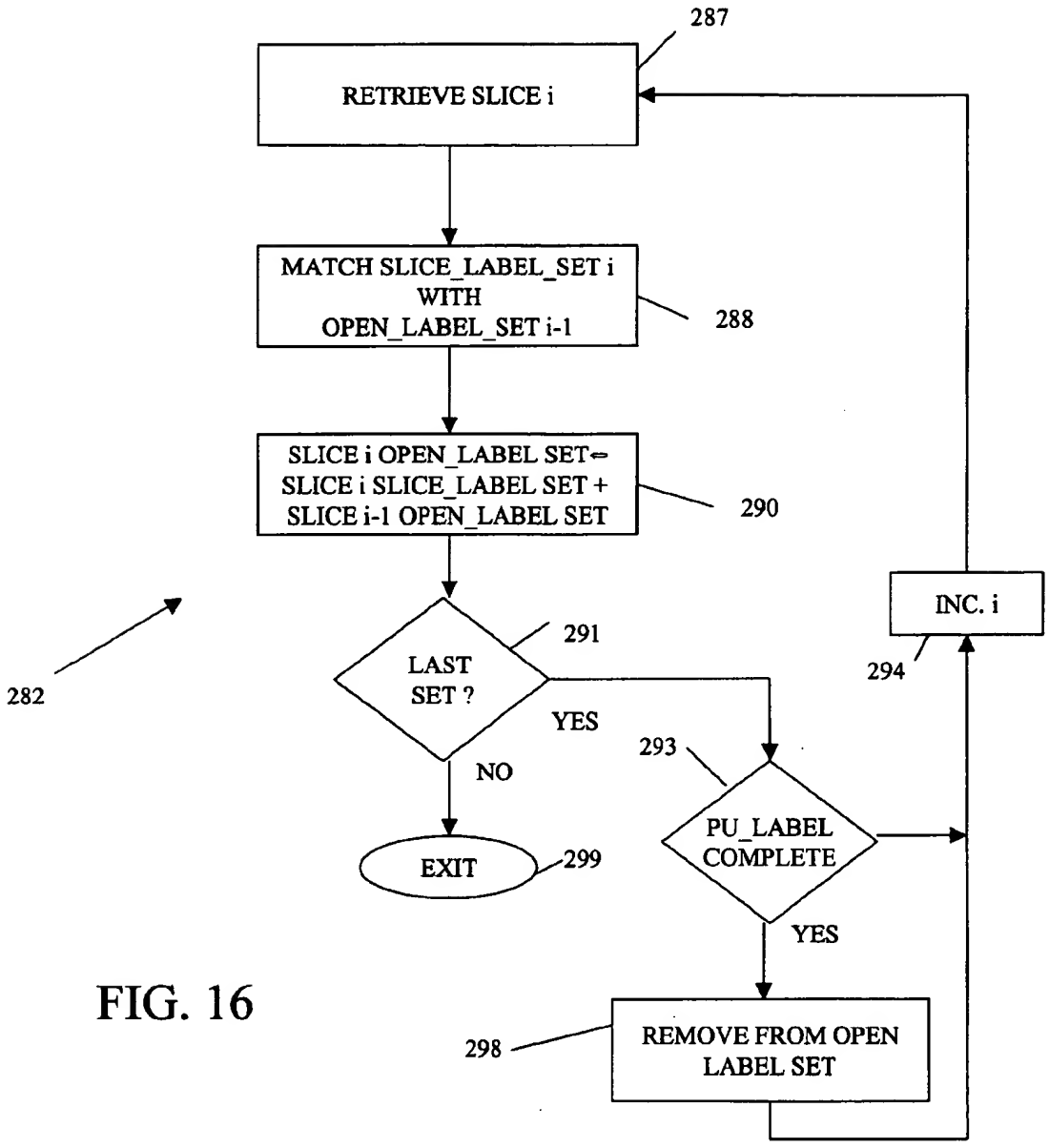

287
RETRIEVE SLICE i
MATCH SLICE_LABEL_SET i WITH OPEN_LABEL_SET i-1
288
SLICE i OPEN_LABEL SET⇐ SLICE i SLICE_LABEL SET + SLICE i-1 OPEN_LABEL SET
290
282
LAST SET ?
291
YES
NO
EXIT
299
293
PU_LABEL COMPLETE
YES
298
REMOVE FROM OPEN LABEL SET
INC. i
294

FIG. 16

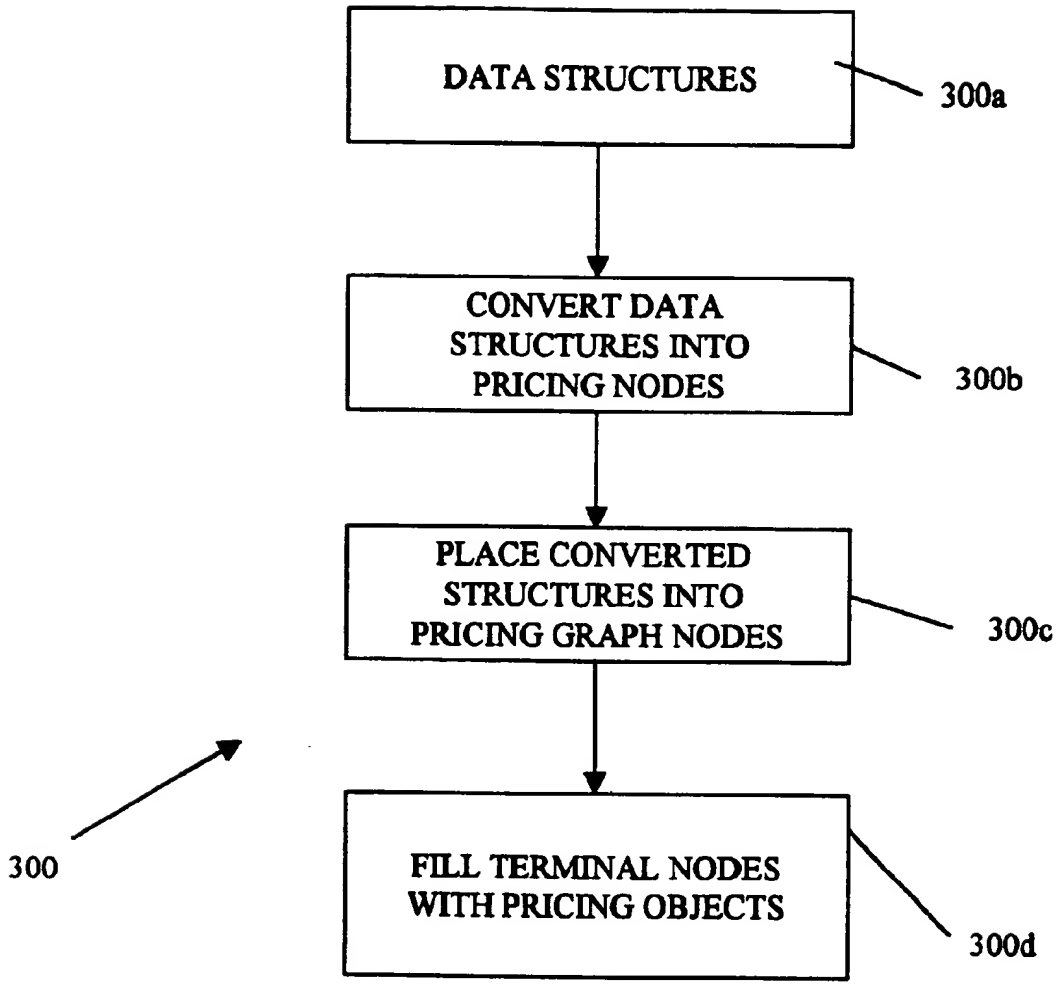

DATA STRUCTURES
300a
CONVERT DATA STRUCTURES INTO PRICING NODES
300b
PLACE CONVERTED STRUCTURES INTO PRICING GRAPH NODES
300c
FILL TERMINAL NODES WITH PRICING OBJECTS
300d
300

FIG.17

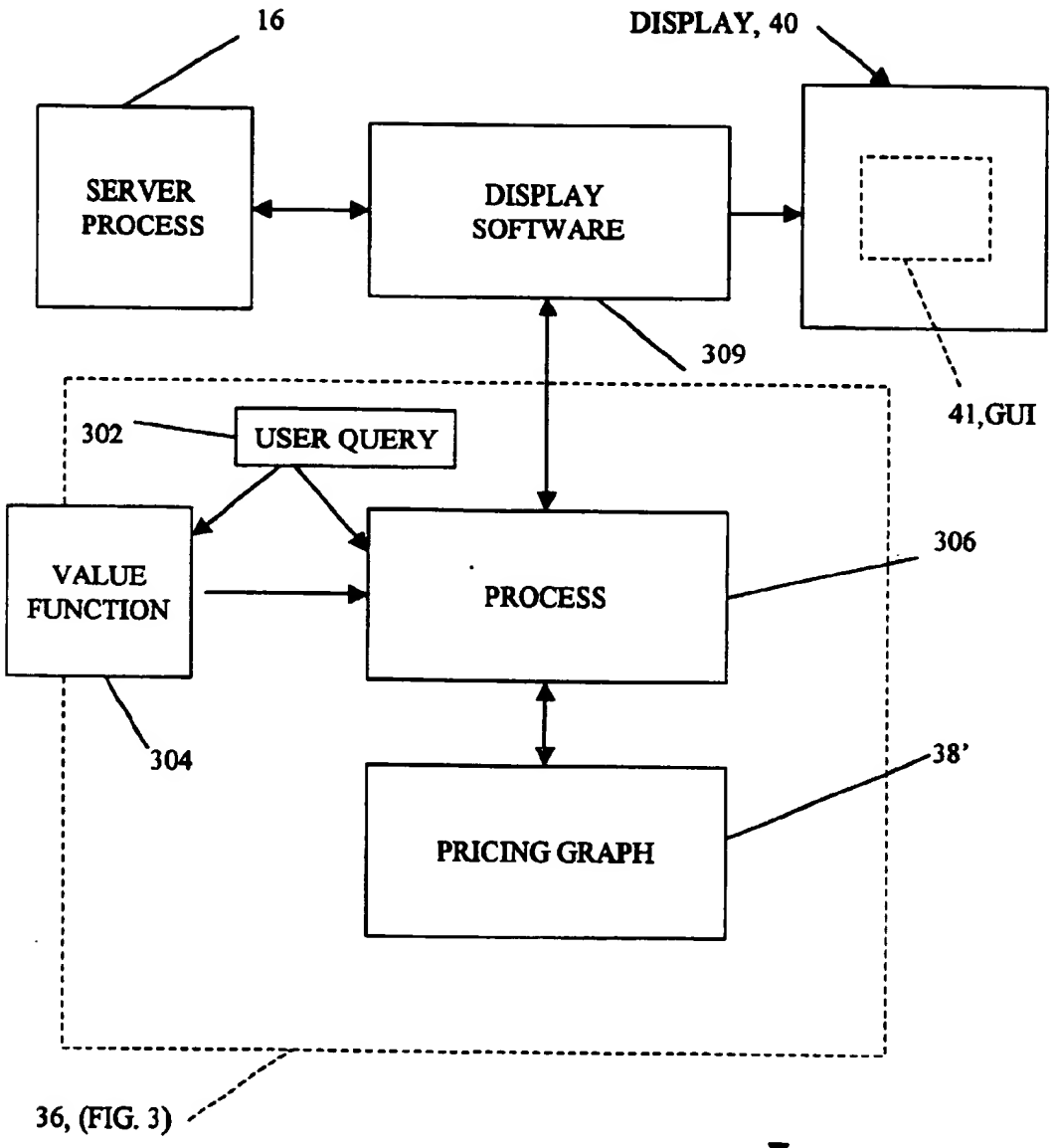

16
DISPLAY, 40
SERVER PROCESS
DISPLAY SOFTWARE
309
41,GUI
302
USER QUERY
VALUE FUNCTION
PROCESS
306
304
38'
PRICING GRAPH
36, (FIG. 3)
301

FIG. 18

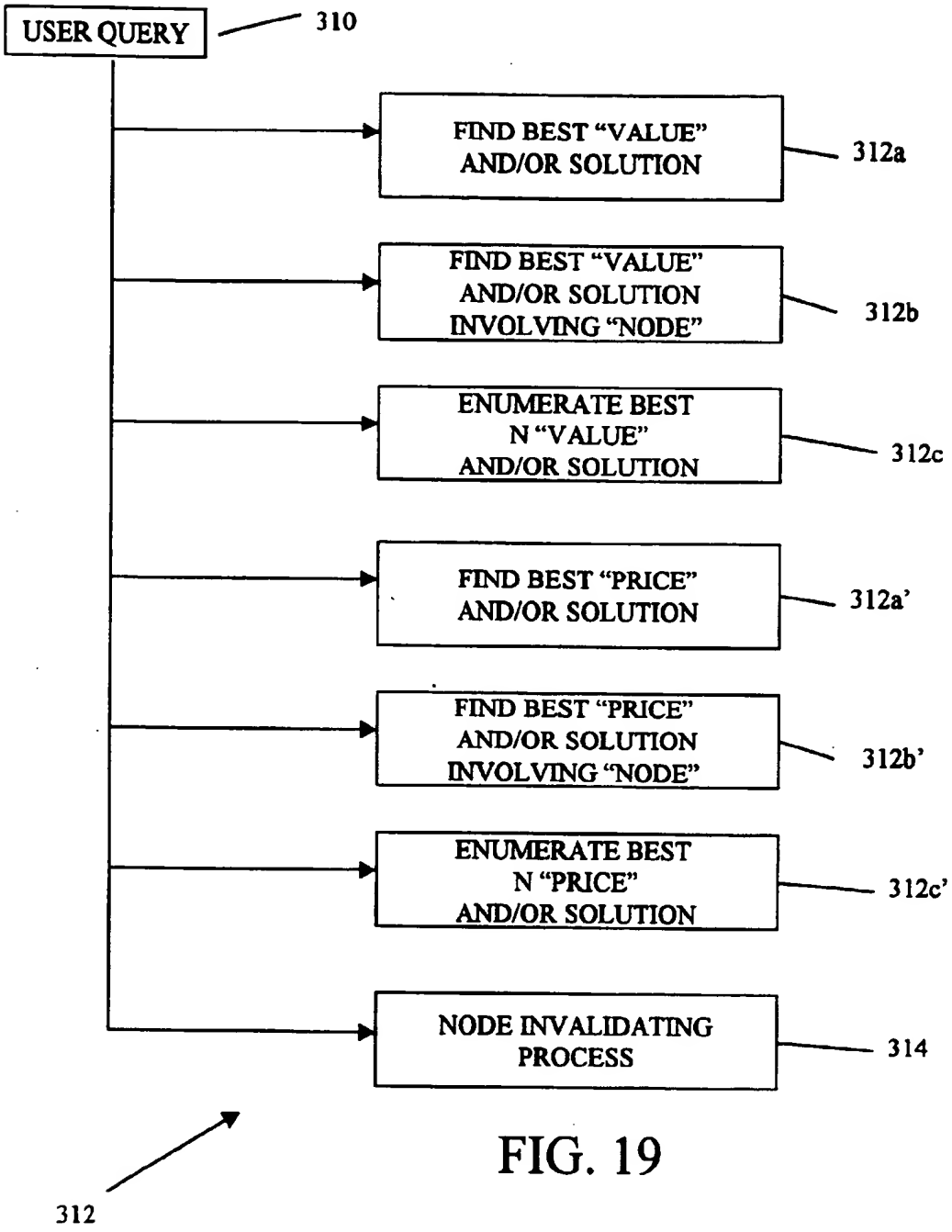
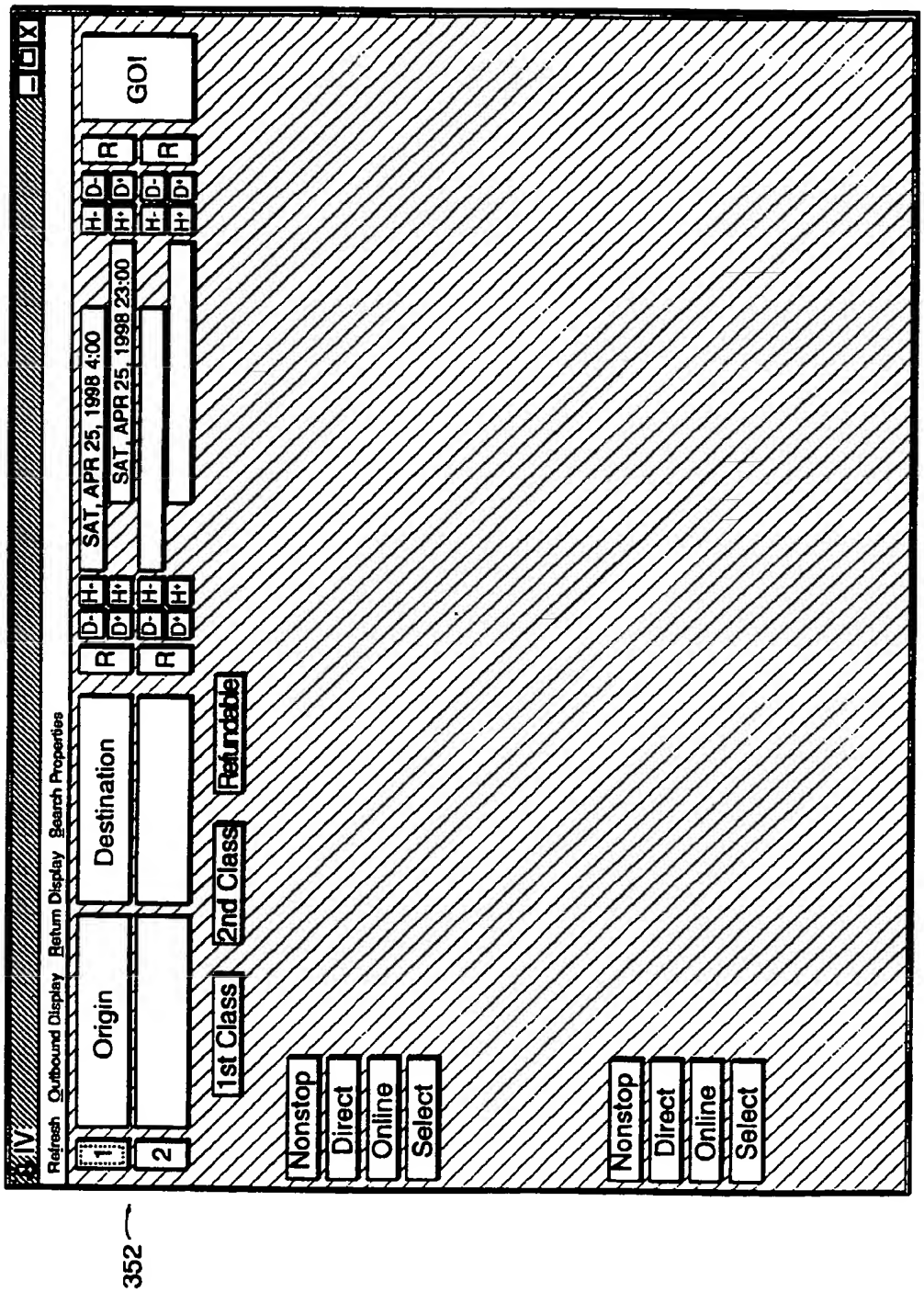

USER QUERY
310
FIND BEST "VALUE" AND/OR SOLUTION
312a
FIND BEST "VALUE" AND/OR SOLUTION INVOLVING "NODE"
312b
ENUMERATE BEST N "VALUE" AND/OR SOLUTION
312c
FIND BEST "PRICE" AND/OR SOLUTION
312a'
FIND BEST "PRICE" AND/OR SOLUTION INVOLVING "NODE"
312b'
ENUMERATE BEST N "PRICE" AND/OR SOLUTION
312c'
NODE INVALIDATING PROCESS
314
312

FIG. 19

FIG. 20


Refresh
Outbound Display
Return Display
Search Properties
Origin
Destination
GO!
SAT, APR 25, 1998 4:00
SAT, APR 25, 1998 23:00
1st Class
2nd Class
Refundable
Nonstop
Direct
Online
Select
Nonstop
Direct
Online
Select
350
352

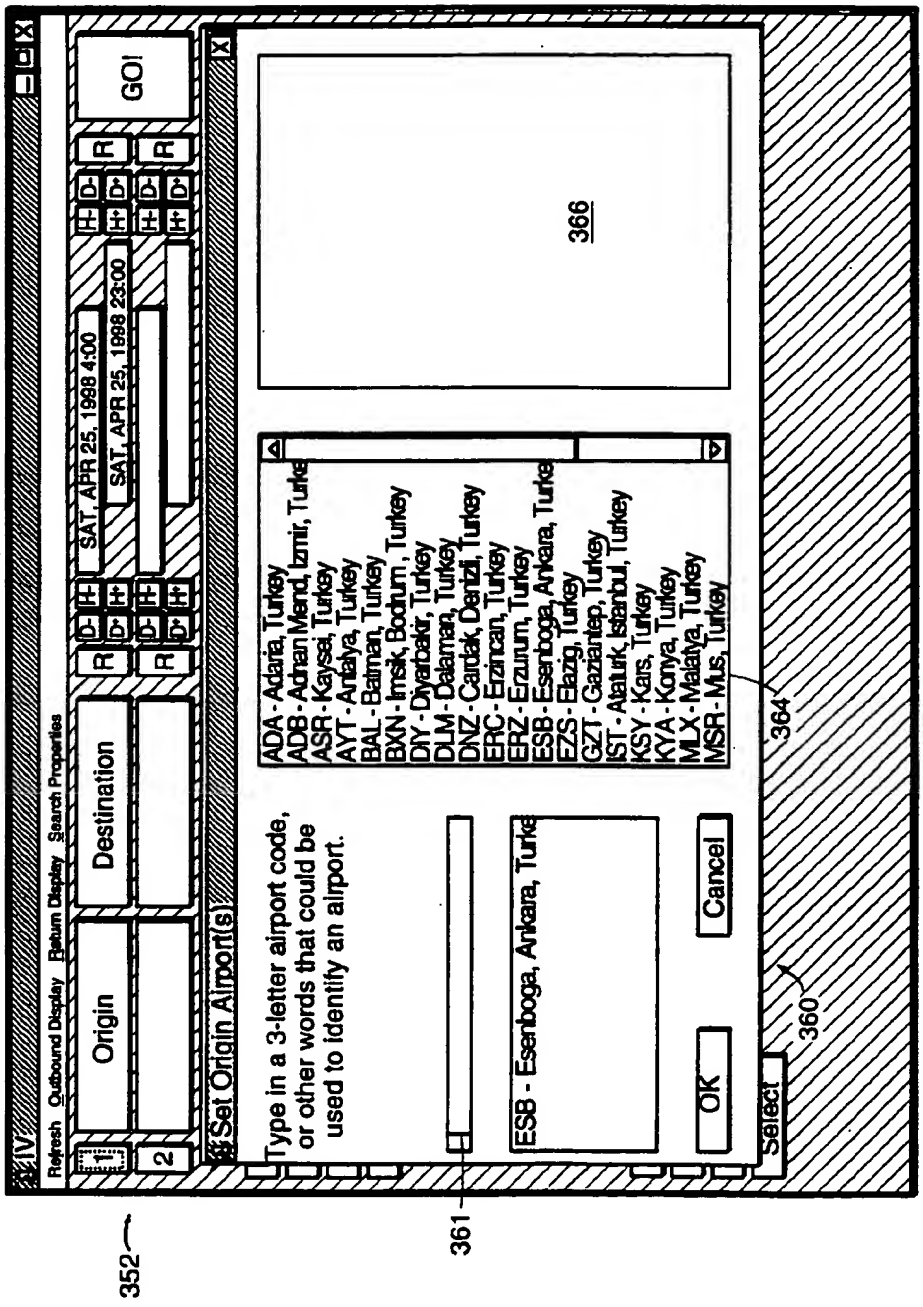

Refresh Outbound Display Return Display Search Properties
Origin
Destination
GO!
SAT, APR 25, 1998 4:00
SAT, APR 25, 1998 23:00
Set Origin Airport(s)
Type in a 3-letter airport code, or other words that could be used to identify an airport.
ADA - Adana, Turkey
ADB - Adnan Mend, Izmir, Turke
ASR - Kayseri, Turkey
AYT - Antalya, Turkey
BAL - Batman, Turkey
BXN - Imsik, Bodrum, Turkey
DIY - Diyarbakir, Turkey
DLM - Dalaman, Turkey
DNZ - Cardak, Denizli, Turkey
ERC - Erzincan, Turkey
ERZ - Erzurum, Turkey
ESB - Esenboga, Ankara, Turke
EZS - Elazig, Turkey
GZT - Gaziantep, Turkey
IST - Ataturk, Istanbul, Turkey
KSY - Kars, Turkey
KYA - Konya, Turkey
MLX - Malatya, Turkey
MSR - Mus, Turkey
ESB - Esenboga, Ankara, Turke
OK
Cancel
Select
366
364
360
361
352
350

FIG. 21

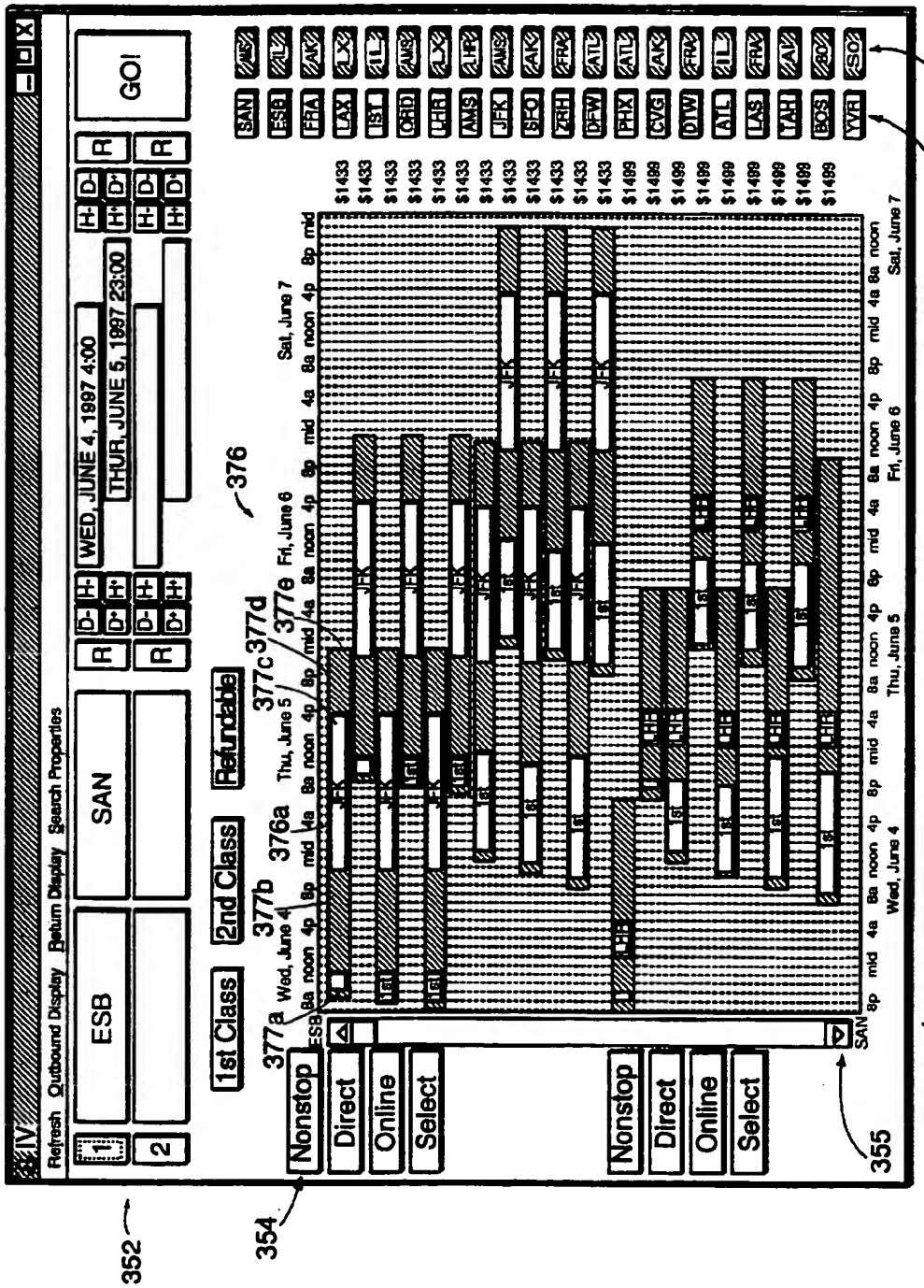

Refresh Outbound Display Return Display Search Properties
ESB
SAN
WED, JUNE 4, 1997 4:00
THUR, JUNE 5, 1997 23:00
GO!
1st Class
2nd Class
Refundable
Nonstop
Direct
Online
Select
352
354
355
370
372
374
376
377a
377b
377c
377d
377e
376a

FIG. 22

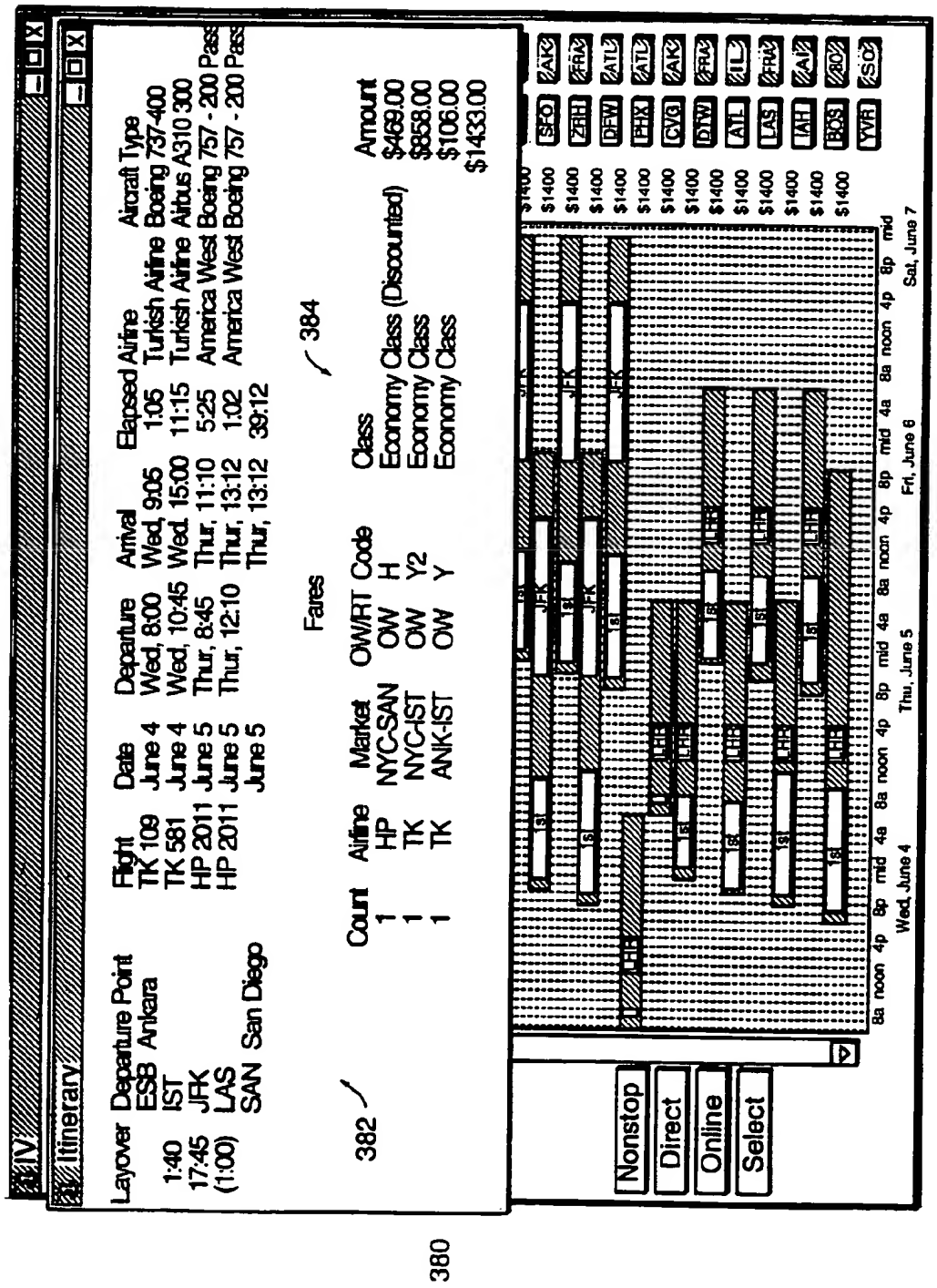

Itinerary
Layover Departure Point Flight Date Departure Arrival Elapsed Airline Aircraft Type
ESB Ankara TK 109 June 4 Wed, 8:00 Wed, 9:05 1:05 Turkish Airline Boeing 737-400
1:40 IST TK 581 June 4 Wed, 10:45 Wed, 15:00 11:15 Turkish Airline Airbus A310 300
17:45 JFK HP 2011 June 5 Thur, 8:45 Thur, 11:10 5:25 America West Boeing 757 - 200 Pass
(1:00) LAS HP 2011 June 5 Thur, 12:10 Thur, 13:12 1:02 America West Boeing 757 - 200 Pass
SAN San Diego June 5 Thur, 13:12 39:12
382
Fares
384
Count Airline Market OW/RT Code Class Amount
1 HP NYC-SAN OW H Economy Class (Discounted) $469.00
1 TK NYC-IST OW Y2 Economy Class $858.00
1 TK ANK-IST OW Y Economy Class $106.00
$1433.00
Nonstop
Direct
Online
Select
380

FIG. 23

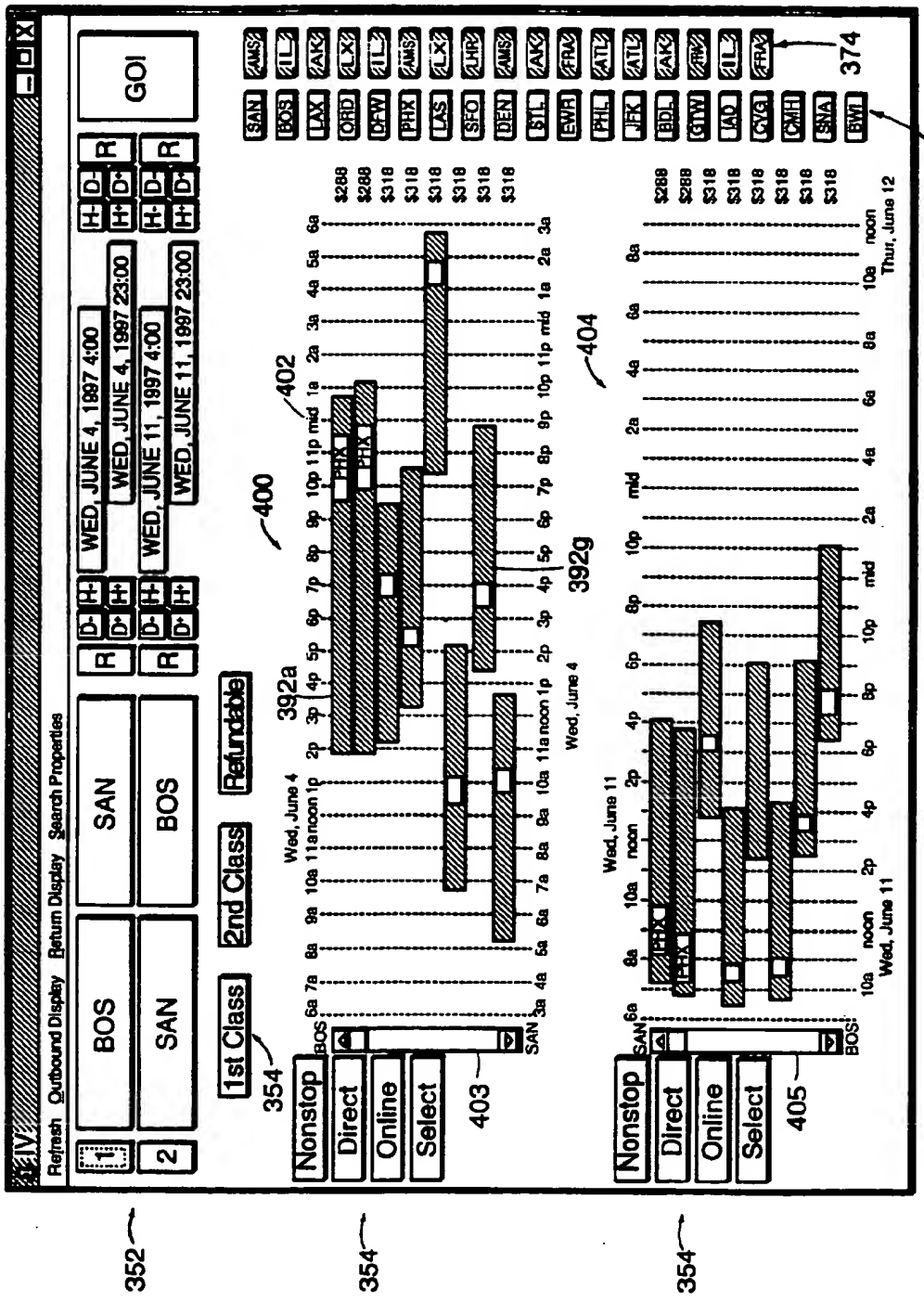

Refresh Outbound Display Return Display Search Properties
BOS
SAN
SAN
BOS
GO!
WED, JUNE 4, 1997 4:00
WED, JUNE 4, 1997 23:00
WED, JUNE 11, 1997 4:00
WED, JUNE 11, 1997 23:00
1st Class
2nd Class
Refundable
Nonstop
Direct
Online
Select
400
402
404
392a
392g
354
403
405
372
374
390
352
354
354

FIG. 24

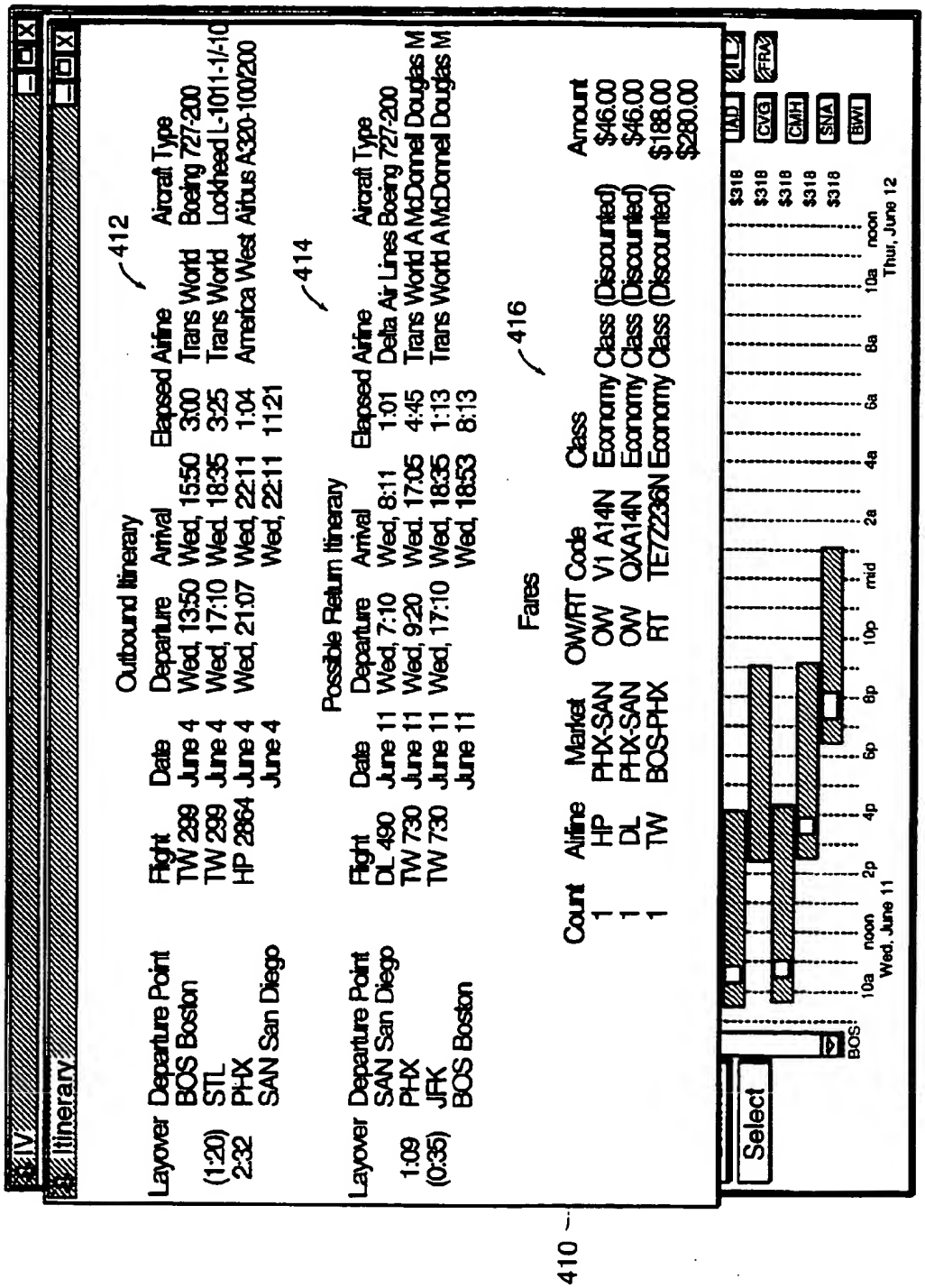

Itinerary
Outbound Itinerary
412
Layover Departure Point Flight Date Departure Arrival Elapsed Airline Aircraft Type
BOS Boston
(1:20) STL TW 289 June 4 Wed, 13:50 Wed, 15:50 3:00 Trans World Boeing 727-200
TW 299 June 4 Wed, 17:10 Wed, 18:35 3:25 Trans World Lockheed L-1011-1/-10
2:32 PHX HP 2864 June 4 Wed, 21:07 Wed, 22:11 1:04 America West Airbus A320-100/200
SAN San Diego June 4 Wed, 22:11 11:21
414
Possible Return Itinerary
Layover Departure Point Flight Date Departure Arrival Elapsed Airline Aircraft Type
SAN San Diego DL 490 June 11 Wed, 7:10 Wed, 8:11 1:01 Delta Air Lines Boeing 727-200
1:09 PHX TW 730 June 11 Wed, 9:20 Wed, 17:05 4:45 Trans World A McDonnel Douglas M
(0:35) JFK TW 730 June 11 Wed, 17:10 Wed, 18:35 1:13 Trans World A McDonnel Douglas M
BOS Boston June 11 Wed, 18:53 8:13
416
Fares
Count Airline Market OW/RT Code Class Amount
1 HP PHX-SAN OW V1 A14N Economy Class (Discounted) $46.00
1 DL PHX-SAN OW QXA14N Economy Class (Discounted) $46.00
1 TW BOS-PHX RT TE7Z236N Economy Class (Discounted) $188.00
$280.00
Select
BOS
10a noon 2p 4p 6p 8p 10p mid 2a 4a 6a 8a 10a noon
Wed, June 11 Thur, June 12
$318 $318 $318 $318 $318
IAD CVG CMH SNA BWI
410

FIG. 25

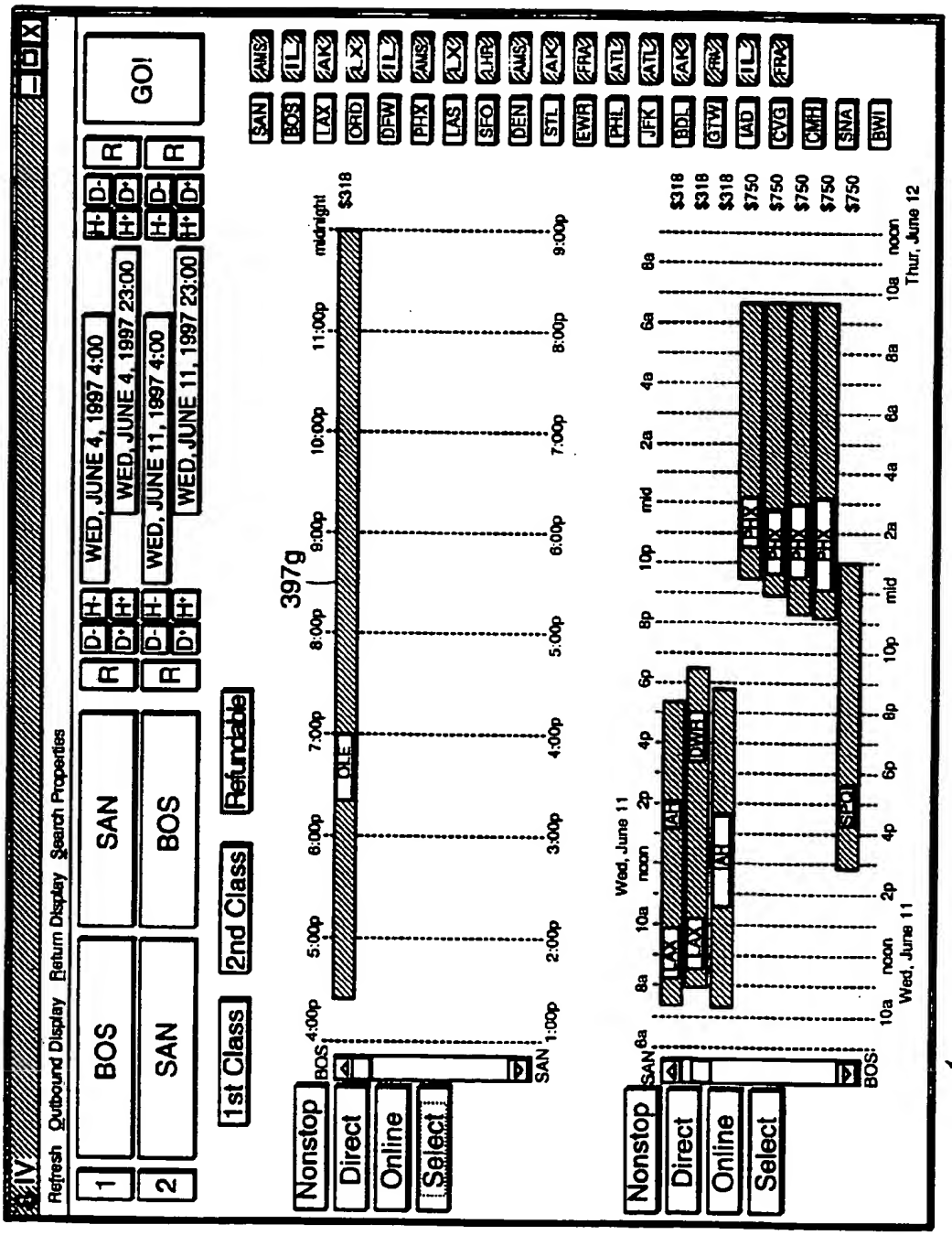

Refresh Outbound Display Return Display Search Properties
1
2
BOS
SAN
SAN
BOS
R
D-
H-
D+
H+
WED, JUNE 4, 1997 4:00
WED, JUNE 4, 1997 23:00
WED, JUNE 11, 1997 4:00
WED, JUNE 11, 1997 23:00
GO!
1st Class
2nd Class
Refundable
Nonstop
Direct
Online
Select
397g
midnight
$318
$750
Wed, June 11
Thur, June 12
350'

FIG. 26

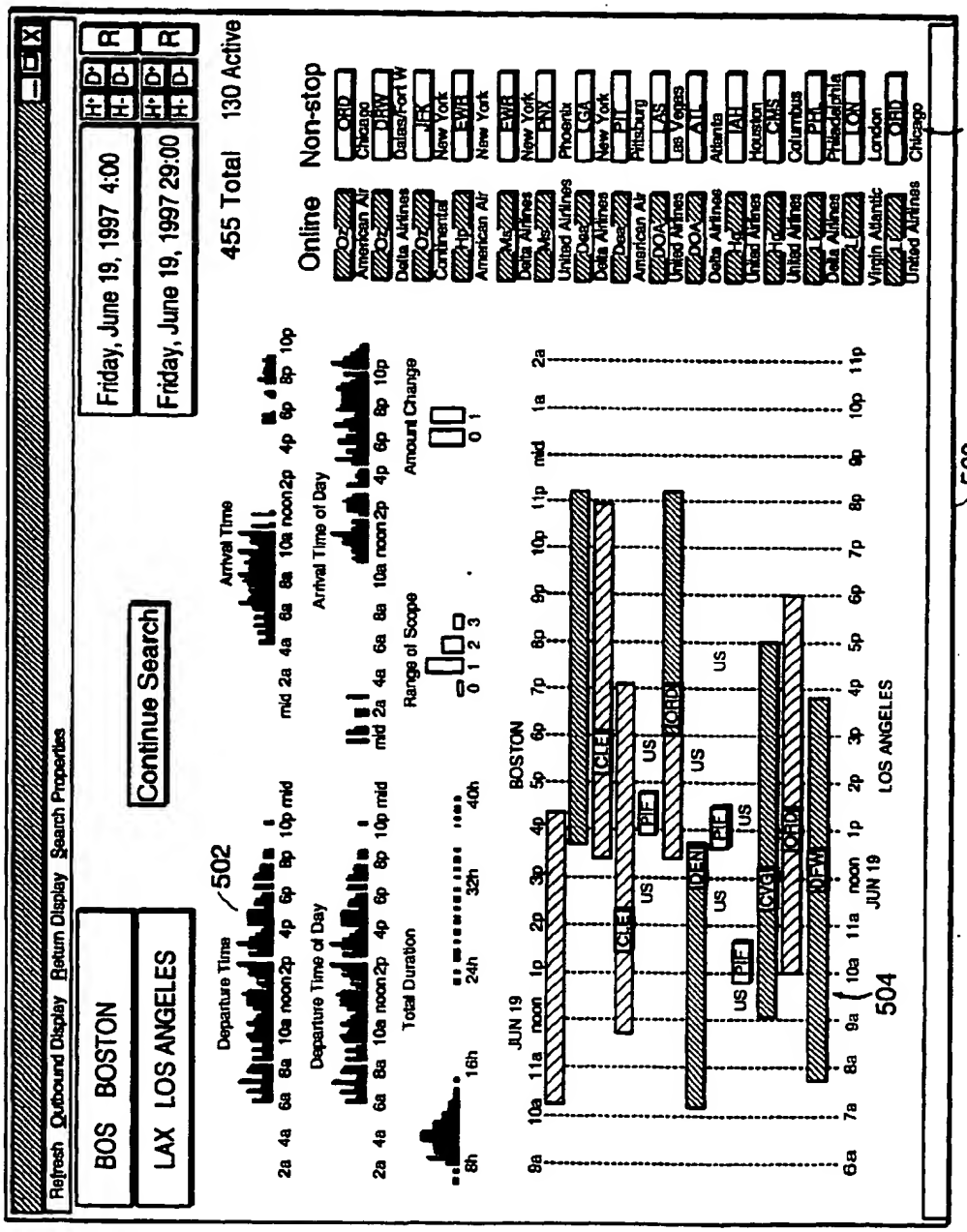

Refresh Outbound Display Return Display Search Properties
BOS BOSTON
LAX LOS ANGELES
Continue Search
Friday, June 19, 1997 4:00
Friday, June 19, 1997 29:00
455 Total 130 Active
Online
Non-stop
Departure Time
Arrival Time
Departure Time of Day
Arrival Time of Day
Total Duration
Range of Scope
Amount Change
BOSTON
LOS ANGELES
JUN 19
502
504
506
500

FIG. 27

TRAVEL PLANNING SYSTEM

BACKGROUND

This invention relates to computerized travel planning systems.

Travel planning systems are used to produce itineraries and prices by selecting suitable travel units from databases containing geographic, scheduling and pricing information. In the airline industry, fundamental travel units include "flights" (sequences of regularly scheduled takeoffs and landings assigned a common identifier) and "fares" (prices published by airlines for travel between two points). The term "itinerary" is often used to refer to a sequence of flights on particular dates, and the term "pricing solution" is often used to refer to a combination of fares and itineraries that satisfies a travel request.

The databases usually contain schedule information provided by airlines, typically in the so-called Standard Schedules Information Manual (SSIM) format, and usually fares published by airlines and resellers, typically provided through the intermediary Airline Tariff Publishing Company (ATPCO). The database may also contain "availability" information that determines whether space is available on flights, or this may be obtained through communication links to external sources such as airlines.

Presently, so-called computer reservation system (CRSs) operate to produce fare and schedule information. There are four generally known computer reservation systems that operate in the United States, Sabre®, Galileo®, Amadeus® and WorldSpan®. The typical CRS contains a periodically updated central database that is accessed by subscribers such as travel agents through computer terminals. The subscribers use the computer reservation system to determine what airline flights are operating in a given market, what fares are offered and whether seats are available on flights to make bookings and issue tickets to clients.

The computer reservation systems typically conduct searches using the information contained in the database to produce itineraries that satisfy a received request. The search results are sorted and returned to the requester's computer for display.

Typically, the number of possible itineraries and pricing solutions that are returned by a CRS is a small portion of the total set that may satisfy a passengers request.

SUMMARY OF THE INVENTION

According to one aspect of the invention, a travel planning system includes a server process that determines travel planning information in response to travel request information and a client process that receives said travel planning information. The client process includes a manipulation process that operates on the travel planning information. In one embodiment, the manipulation process can operate on the travel planning information in accordance with at least one user preference input.

manipulation

According to another aspect of the invention, an airline travel planning system includes a scheduler that produces a set of flights in response to a user specified query, a faring system that provides fares for the sets of flights, and which represents the sets of flights and fares as a set of logically manipulatable nodes in a data structure. The system also includes an enumeration process that processes the data structure to extract flight-fare components from nodes in the data structure.

According to a still further aspect of the invention, a travel planning system includes a server process that in response to at least one travel destination and at least one travel origin determines a set of pricing solutions, said set of pricing solutions represented by a structure that contains a plurality of logically combinable entries that represent a second plurality of pricing solution entities and a client process that receives said pricing solution structure. The client process includes an enumeration process that extracts pricing solutions from the structure.

According to a still further aspect of the invention, a travel planning system includes a server process that in response to at least one travel destination and at least one travel origin determines and represents a set of pricing solutions by a directed acyclic graph data structure and a client process that receives said directed acyclic graph. The client process uses the directed acyclic graph representation to enumerate in response to user preferences a set of pricing solutions.

According to a still further aspect of the invention, an airline travel planning system includes a server computer and a server process executed on said server computer, said server process including a search process to search for set of pricing solutions that determine possible set of pricing solutions in accordance with at least one destination and at least one origin, said search process representing said set of pricing solutions in the form of a directed acyclic graph and a client computer and a client computer process responsive to the set of pricing solutions represented in the form of the directed acyclic graph. The client process including a manipulation process that manipulates the set of pricing solutions in the form of the directed acyclic graph representation in response to user preferences. The manipulation process includes a pruning process responsive to user preferences that alters the directed acyclic graph representation in such a manner so as to eliminate undesirable pricing solutions, and an enumeration process responsive to user preferences that produces a sorted subset of the pricing solutions represented in the directed acyclic graph.

One or more advantages are provided by the some of the aspects of the invention. The client process receives a set of pricing solutions provided in a compact representation. A preferred, compact representation of the set of pricing solutions is as a data structure comprising a plurality of nodes that can be logically manipulated using value functions to enumerate a set of pricing solutions. One preferred example is a graph data structure type particularly a directed acyclic graph that contains nodes that can be logically manipulated or combined to extract a plurality of pricing solutions. The client, can store and/or logically manipulate the set of pricing solutions to extract or display a subset of the set of pricing solutions without the need for additional intervention by the server.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing features and other aspects of the invention will be described in further detail by the accompanying drawings, in which:

FIG. 1 is a block diagram of a client server travel planning system.

FIG. 2 is a flow chart showing a server process used in the system of FIG. 1.

FIG. 3 is a flow chart showing a client process used in the system of FIG. 1.

FIGS. 3A–3B are diagrammatic representations of pricing graphs.

FIGS. 4A–4B are flow charts showing a faring process used in the server process of FIG. 2.

FIG. 5 is a flow chart showing a process to decompose an itinerary into faring atoms used in the process of FIG. 4A.

FIG. 6 is a flow chart showing a process for partitioning itineraries used in the process of FIG. 5.

FIG. 7 is a flow chart showing a process for applying rules to faring atoms used in the faring process of FIG. 4A.

FIGS. 8A–8C are flow charts showing a process for retrieving fare rules.

FIG. 9 is a flow chart showing a process for applying fare rules.

FIG. 10 is a flow chart showing a process for determining priceable units used in the faring process of FIGS. 4A–4B.

FIG. 10A is a flow chart showing a process for enumerating collections of sets of faring components used in the process of FIG. 10.

FIG. 11 is a flow chart showing a process for enumerating collections of faring components used in the process of FIG. 10.

FIG. 12 is a flow chart showing a process for representing priceable units in a compact representation.

FIGS. 13–15 are flow charts showing processes for determining priceable units.

FIG. 16 is a flow chart showing a process for producing slice label sets.

FIG. 17 is a flow chart showing the process for constructing a pricing graph.

FIG. 18 is a block diagram showing the relationship between the pricing graph and a graphical user interface for the travel planning system of FIG. 1.

FIG. 19 is a flow chart showing various enumeration functions.

FIG. 20 is a diagram of a window depicting a user interface.

FIG. 21 is a diagram of a window used in an initial query;

FIG. 22 is a diagram of a window depicting an exemplary solution of a one-way travel request.

FIG. 23 is a diagram of a window depicting an itinerary and associated fares corresponding to one of the solutions depicted in FIG. 21.

FIG. 24 is a diagram of a window depicting an exemplary solution of round trip travel request.

FIG. 25 is a diagram of a window depicting an outbound itinerary and a possible return itinerary and associated fares corresponding to one of the solutions depicted in FIG. 24.

FIG. 26 shows the window generally depicted in conjunction with FIG. 24 modified based upon a user selected criteria.

FIG. 27 shows a window depicting a histogram.

DETAILED DESCRIPTION

Referring now to FIG. 1, a travel planning system 10 is shown. The travel planning system can be used with various forms of travel such as airline, bus and railroad and is particularly adapted for air travel. It includes a server computer 12 having a computer memory or storage media 14 storing a server process 15. The server process includes a scheduler process 16 and a faring process 18. The scheduler process 16 is any suitable scheduler process that will produce from a travel request sets of flights that can satisfy the request. The faring process 18 is a process that determines a set of valid fares and links the set of valid fares to the sets of flights to form a pricing solution. The server process 15 can be configured to produce other travel-related information as a result of a user query. For example, the server process 12 can produce routes or airline suggestions, optimal travel times and suggestions for alternative requests.

The travel planning system 10 also includes a plurality of databases 20*a*, 20*b* which store industry-standard information pertaining to travel (e.g., airline, bus, railroad, etc.). For example, database 20*a* can store the Airline Tariff Publishing Company database of published airline fares and their associated rules, routings and other provisions, the so-called ATPCO database. Database 20*b* can be an inventory of current availability of airline information for a particular carrier and so forth. The databases 20*a*–20*b* are typically stored locally and updated periodically by accessing remote resources 21*a*, 21*b* that maintain the respective databases.

The system 10 also includes a plurality of clients 30*a*–30*c* (generally 30) implemented by terminals or preferably personal computers. The clients 30*a*–30*c* are coupled to the server 12 via a network 22 which is also used to couple the remote resources (21*a*–21*c*) that supply the databases 20*a*–20*b* to the server 12. The network 22 can be any local or wide area network or an arrangement such as the Internet.

The clients 30*a*–30*c* are preferably smart clients. That is, using client 30*c* as an illustrative example, client 30*c* includes a client computer system 32 including a computer memory or storage media 34 that stores a client process 36 and a set of pricing solutions 38. The set of pricing solutions 38 in one embodiment is provided from the server process 15 and comprises a set of fares that are valid for a journey, and associated information linking the fares to the flight segments of the journey.

The set of pricing solutions 38 is obtained from the server 12 in response to a user request sent from the client 30c to the server 12. The server 12 executes the server process 15 using the scheduling process 16 and the faring process 18 to produce a set of pricing solutions for a particular journey. If requested by the client, for example client 30*c*, the server 12 will deliver the set of pricing solutions 38 to the requesting client 30*c*. Under control of the client process 36, the requesting client 30*c* can store and/or logically manipulate the set of pricing solutions 38 to extract or display a subset of the set of pricing solutions as a display representation 41 on the monitor 40.

SERVER PROCESS

Referring now to FIG. 2, the server process 18 is preferably executed on the server computer 12 but could be executed on the client computer 32. The server process 18 is responsive to a user input query 48. The user input query 48 would typically include minimal information needed to determine a set of pricing solutions. This information typically requires at a minimum, an origin and a destination for travel. In addition, the information could also include times, dates and so forth.

This query 48 is fed to the scheduler process 16 that produces a large number of itineraries, that is, sequences of flight segments between the origin and destination for each slice of a journey. Examples of scheduler systems that may be used include the OAG Flight Desk (Official Airlines Guide, a division of Reed Travel Group) or schedule components of computer reservation systems (CRS's) such as Sabre®, Apollo®, Amadeus® and WorldSpan®. It is preferable in order to obtain the largest number of possible itineraries to use a scheduler with dynamic connection generation. Such a scheduler is described in co-pending patent application entitled SCHEDULER SYSTEM FOR TRAVEL PLANNING SYSTEM, Ser. No. 09/109,622, filed on Jul. 2, 1998 by Carl G. deMarcken et al. pending and assigned to the assignee of the invention and incorporated herein by reference.

The scheduler process **16** provides the itineraries to a faring process **18**. The faring process **18** provides a set of pricing solutions **38** by finding valid fares corresponding to the itineraries produced by the scheduler process **16**. The faring process **18** validates the fares for inclusion in the set of pricing solutions **38**.

The set of pricing solutions **38** is used by an availability system **58** that interrogates an airline inventory database **20***b* to determine whether there are seats available on particular flights for particular pricing solutions. The availability system **58** uses the airline inventory database **20***b* as a filter to remove from the set of pricing solutions **38** those pricing solutions for which there are not available seats. The availability system **58** is shown after the faring process **18**. However, it could be included at nearly any point in the server process **18**. In addition, it is shown being fed by the pricing solution when it may only receive flight information from the scheduler process **16** depending on the airline.

The client system **30***c* receives the results from the server process **18**. These results are the set of pricing solutions **38** and/or pricing solutions based upon availability. The client process **36** executed in the client **30***c* uses this information or a subset of it to access a booking system **62** to provide a booking and reservation for a user selected, enumerated pricing solution, as will be described below.

CLIENT PROCESS

Referring now to FIG. **3**, the client process **36** receives a listing of possible itineraries from the scheduler process **16** as well as the set of fares from the faring process **18** or the availability system **58**. The set of pricing solutions **38**, if obtained from the faring process **18**, will include a large number of pricing solutions for which there is not any available inventory. Therefore, the components would need to be first checked out with an airline prior to booking. The set of pricing solutions **38** if obtained after the availability system **58** should contain pricing solutions which have a high degree of availability for booking on an airline.

In one embodiment, the set of pricing solutions **38** is provided in a compact representation **38'**. A preferred, compact representation **38'** of the set of pricing solutions **38** is as a data structure comprising a plurality of nodes including itineraries and fares and that can be logically manipulated using value functions to enumerate a set of pricing solutions. One preferred example is a graph data structure type particularly a directed acyclic graph (DAG) that contains nodes that can be logically manipulated or combined **74** to extract **75** a plurality of pricing solutions for display **72**.

The client process **36** receives the flight information from scheduler process **16** and the pricing solution from the faring process **18** or the availability system **56** and enumerates pricing solutions from the directed acyclic graph (DAG) representation. The enumerated set **70** of pricing solutions is rendered or displayed **72** in a graphical user interface **41** on the client monitor **40** (FIG. **1**) in a manner as will be described below.

In response to user input **76**, the client **40** can manipulate **74** travel options and can query the local copy of the DAG to produce and display a subset of pricing solutions enumerated from the DAG that satisfy the query **76**. The manipulation process used to control the display and change the travel options will be described below.

A directed acyclic graph (DAG) is used to represent the compact set of pricing solutions **38'** since, in general, the number of nodes needed to represent a typical pricing solution will be substantially less than the actual number of pricing solutions represented by the DAG. This significantly increases the efficiency of transfer of a set of pricing solutions **38** from the server process **18** to the client process **36**. The DAG representation also minimizes the storage requirements for the set of pricing solutions **38**. The DAG representation permits the use of powerful search, sorting and manipulation processes to produce various subsets of set of pricing solutions in an efficient manner. As used herein, a directed acyclic graph (DAG) is a set of nodes connected by directed arcs, that have no loops of arcs in the same direction. If a node A is connected to a node B via an arc A-B, then A is called a parent of B, and B is called a child of A. Each node may have zero, one or many parents and zero, one or many children. As used herein, a pricing solution that is represented by a graph will be referred to as a pricing graph.

PRICING-GRAPH

A pricing graph that is produced by the faring process **18** and that represents a pricing solution includes three types of nodes. The first type of node is an exclusive node, i.e., "OR" node. An OR node N with children A, B and C represents an exclusive choice between A, B and C. In other words, a pricing-solution involving node N contains either the fares and itineraries represented by A, or by B, or by C.

The second type of node is a collection node, i.e., an "AND" node. An AND node N with children A, B and C represents the sum of A, B and C. In other words, a pricing solution involving N contains all the fares and itineraries found within A, B and C.

The third type of node is a terminal node. Terminal nodes are used to hold pricing objects. Pricing objects include fares, itineraries, surcharges, routes, prices, booking codes, taxes, rules/restrictions and other information of the user or information that might be part of a travel option. Collectively, "AND" and "OR" nodes are non-terminal nodes.

An example of the pricing-graph for a hypothetical round-trip journey is presented below in TABLE 1. For each node, its type and children are listed. If a node is a terminal, the fare or itinerary is provided. Many nodes in the pricing graph have more than one parent.

TABLE 1

Node	Type	Children	Object
0	OR	Nodes 1, 2, 3	
1	AND	Nodes 10, 14, 17, 17	
2	AND	Nodes 4, 5	
3	AND	Nodes 13, 15, 19, 19	
4	OR	Nodes 8, 9	
5	OR	Nodes 6, 7	
6	AND	Nodes 14, 16	
7	AND	Nodes 15, 18	
8	AND	Nodes 13, 16	

TABLE 1-continued

Node	Type	Children	Object
9	AND	Nodes 13, 18	
10	OR	Nodes 11, 12	
11	Itin.		Slice 1: BOS→LAX UA023
12	Itin.		Slice 1: BOS→DFW UA100, DFW→LAX UA103
13	Itin.		Slice 1: BOS→SAN NW222
14	Itin.		Slice 2: LAX→BOS UA515
15	Itin.		Slice 2: SAN→BOS NW223
16	Fare		UA BOS-LAX One-way "Y"
17	Fare		UA BOS-LAX Round-trip "QE7NR"
18	Fare		NW BOS-SAN One-way "F"
19	Fare		NW BOS-SAN Round-trip "H7NR"

This pricing-graph represents a total of nine pricing solutions. These solutions can be extracted from the pricing-graph by descending from the root node, node 0. At every OR node a choice between children is made, and the choice determines the pricing-solution that results. At every AND node each child branch is descended, and the results are combined.

The term BOS→LAX UA023 is an itinerary which uses standard nomenclature to represent airports BOS and LAX, airline UA, and flight number 023. In general, conventional nomenclature used in the airline industry will be used herein.

The set of pricing-solutions that represented in the pricing-graph is presented in TABLE 2 below.

TABLE 2

Solution Number	Itineraries	Fares
1	Slice 1: BOS→LAX UA023	UA BOS-LAX RT "QE7NR"
	Slice 2: LAX→BOS UA515	UA BOS-LAX RT "QE7NR"
2	Slice 1: BOS→LAX UA023	UA BOS-LAX OW "Y"
	Slice 2: LAX→BOS UA515	UA BOS-LAX OW "Y"
3	Slice 1: BOS→LAX UA023	UA BOS-LAX OW "Y"
	Slice 2: SAN→BOS NW223	NW BOS-SAN OW "F"
4	Slice 1: BOS→DFW UA100, DFW_LAX UA103	UA BOS-LAX RT "QE7NR"
	Slice 2: LAX→BOS UA515	UA BOS-LAX RT "QE7NR"
5	Slice 1: BOS→DFW UA100, CFW_LAX UA103	UA BOS-LAX OW "Y"
	Slice 2: LAX→BOS UA515	UA BOS-LAX OW "Y"
6	Slice 1: BOS→DFW UA100, DFW_LAX UA103	UA BOS-LAX OW "Y"
	Slice 2: SAN→BOS NW223	NW BOS-SAN OW "F"
7	Slice 1: BOS→SAN NW222	NW BOS-SAN OW "F"
	Slice 2: LAX→BOS UA515	UA BOS-LAX OW "Y"
8	Slice 1: BOS→SAN NW222	NW BOS-SAN RT "H7NR"
	Slice 2: SAN→BOS NW223	NW BOS-SAN RT "H7NR"
9	Slice 1: BOS→SAN NW222	NW BOS-SAN OW "F"
	Slice 2: SAN→BOS NW223	NW BOS-SAN OW "F"

The pricing-graph encodes the requirement that two itineraries are combined, one from slice 1 and one from slice 2, to form a pricing solution. Further, each itinerary is spanned by fares. In this case each pricing solution involves two fares, and round-trip fares are combined with like round-trip fares. In most circumstances, the number of nodes in the pricing-graph is small compared to the number of pricing-solutions those nodes represent. In many cases, a graph of 10,000 or so nodes can represent more than 1,000,000,000 pricing-solutions.

Referring now to FIG. 3A, the nodes of the pricing graph corresponding to Table 1 are shown, as an example. This figure illustrates the manner in which nodes in the pricing graph data structure as represented in Table 1 are combined to provide the pricing solutions shown in Table 2. Using pricing solution No. 1 (from TABLE 2) as an example, it can be shown that starting at the top of the graph at node 0, node 0 allows for a choice between nodes 1, 2, and 3. For pricing solution No. 1, Node 1 is chosen. Node 1 is the AND node that points to nodes 10 and 14, and has two pointers to node 17. Node 10 is an OR node which provides a choice of either nodes 11 or nodes 12. Node 11 as shown in FIG. 3A corresponds to a terminal node, the itinerary (BOS-LAX UA 023). Node 12 corresponds to a terminal node, the itinerary BOS-DFN UA 100, DFN-LAX UA 103. This second choice in node 10 will provide pricing solutions corresponding to numbers 4–6, respectively. Therefore, selecting node 11 provides the itinerary for the first slice of solution 1. The fare for pricing solution 1 is provided by node 17 which has two pointers, one for each slice, to the fare "US BOS-LAX RT QE7NR" corresponding to the fare shown for pricing solution no. 1 in Table 2 for the first slice. The second itinerary for pricing solution no. 1 is provided by node 14 which is referenced in AND node 1 that points to the itinerary LAX-BOS UA 515. The corresponding fare is also from terminal node 17 since it is a round trip fare UA BOS-LAX RT QE7NR.

A second one of the pricing solutions, for example, the pricing solution 4 incorporating the terminal node 12 is provided by starting at node 0, and using node 1. Node 1 is an AND node requiring that nodes 17 (twice), node 10, and node 14 be included. Node 10 is an OR node as mentioned above and is used to select node 12 which is the itinerary including segments "BOS→DFW UA 100" and "DFW→LAX UA 103". From node 1, node 14 the return itinerary LAX-BOS UA 515 also is reached. Node 17 also is chosen which contain the round trip fares. Similarly, the

remaining ones of the pricing solutions can be extracted from the pricing graph in the same manner as the two examples given above.

As mentioned above, a graph will typically have many more pricing solutions than nodes in the graph. The example just illustrated in conjunction with FIG. **3A** has 9 pricing solutions and 19 nodes which is an exception to that general rule. Another example of a pricing graph which does satisfy that general observation is shown in conjunction with FIG. **3B**.

Referring now to FIG. **3B**, a pricing graph is shown having 43 nodes N0–N42 that when combined represent 856 pricing solutions. Each node in the pricing graph has a number associated with it corresponding to the number of pricing solutions that is represents. In order to make this illustration of manageable size, identifiers (representing the nodes of the terminals) are substituted in the pricing graph for the actual terminal objects of the graph. Thus, as shown in FIG. **3B**, outbound and return itineraries, and fare nodes are represented by the Nodes N20–N42.

This pricing graph (TABLE 3) has 9 itineraries which can be combined with 14 fares represented by 13 AND nodes and 7 OR nodes. The pricing objects are represented by 23 nodes. The pricing graph has a combined total of 43 nodes to represent 876 pricing solutions.

FIG. **3B** shows examples of a pricing graph for a round trip LAX-BOS journey. This example shown in FIG. **3B** is generally more representative of an outcome of a faring search. That is, generally the pricing graph represents more pricing solutions than nodes contained in the graph.

TABLE 3

Node	Type	Children	Object
0	AND	Nodes 1, 6, 11	
1	OR	Nodes 2, 3, 4	
2	AND	Nodes 5, 40	
3	AND	Nodes 41, 41	
4	AND	Nodes 42, 42	
5	OR	Nodes 39, 40	
6	OR	Nodes 7, 8, 9	
7	AND	Nodes 20, 10	
8	AND	Nodes 21, 10	
9	AND	Nodes 22, 10	
10	OR	Nodes 23, 24, 25, 26	
11	OR	Nodes 12, 13, 14, 16, 17, 18	
12	AND	Nodes 27, 15	
13	AND	Nodes 28, 15	
14	AND	Nodes 29, 15	
15	AND	Nodes 30, 31, 32	
16	AND	Nodes 33, 19	
17	AND	Nodes 34, 19	
18	AND	Nodes 35, 19	
19	OR	Nodes 36, 37, 38	
20	Itin.		Slice 1: LAX→DFW NW100, DFW→BQS AA223
21	Itin.		Slice 1: LAX→DFW NW137, DFW→BOS AA223
22	Itin.		Slice 1: LAX→DFW NW137, DFW→BOS AA414
23	Fare		DFW, LAX NW "Y" OW
24	Fare		DFW, LAX NW "F" OW
25	Fare		DFW, LAX NW "C" OW
26	Fare		DFW, LAX NW "QA7" OW
27	Itin.		Slice 2: BOS→DFW AA67, DFW→LAX CO716
28	Itin.		Slice 2: BOS→DFW AA67, DFW→LAX CO717
29	Itin.		Slice 2: BOS→DFW AA67, DFW→LAX CO719
30	Fare		DFW, LAX CO "F" OW
31	Fare		DFW, LAX CO "C" OW
32	Fare		DFW, LAX CO "Y" OW
33	Itin.		Slice 2: BOS→DFW AA852, DFW→LAX DL186
34	Itin.		Slice 2: BOS→DFW AA852, DFW→LAX DL180
35	Itin.		Slice 2: BOS→DFW AA852, DFW→LAX DL343
36	Fare		DFW, LAX DL "F" OW
37	Fare		DFW, LAX DL "C" OW
38	Fare		DFW, LAX DL "Y" OW
39	Fare		DFW, BOS AA "QE7NR" RT
40	Fare		DFW, BOS AA "QE7IP" RT
41	Fare		DFW, BOS AA "QE14NR" RT
42	Fare		DFW, BOS AA "QE21NR" RT

THE FARING SYSTEM

Referring now to FIGS. **4A** and **4B**, the faring process **18** includes a process **80** to retrieve itinerary sets for all slices in an itinerary. The itinerary sets are provided from the scheduler process **16** for each slice of a journey where a slice corresponds to a direction of travel. Thus, for example, for a round trip journey there would be two slices, one for the outbound part of the journey and one for the return part of the journey. The faring process **18** decomposes **82** the itinerary into faring atoms. As used herein, faring atoms refer to a sequence of flight segments or equivalently legs that are spanned by a single fare. For example, the itinerary

UA005 from DFW to BOS at 12:30 on 12NOV

UA010 from BOS to YYZ at 18:00 on 12NOV

AC121 from YYZ to YVR at 01:00 on 13NOV

permits the following faring-atoms as shown in TABLE 4.

TABLE 4

Faring-Atom Number	Legs and Departure Times
1	DFW→BOS UA005 12NOV 12:30
2	BOS→YYZ UA010 12NOV 18:00
3	DFW→BOS UA005 12NOV 12:30 BOS→YYZ UA011 12NOV 18:00
4	YYZ→YVR AC121 13NOV 01:00

A faring atom is represented by a data structure that preferably includes the following fields as shown in TABLE 5:

TABLE 5

Faring-Atom fields	Use
legs-and-departure-times	A list of legs and their departure times and dates.
faring-market	The faring-market that this faring-atom is in.
cached-results	A storage space used to eliminate redundant computation in the rule-checking process. As rule record-2s are applied to faring-atoms, the results are stored in this field. It the same record-2 is applied again, the answer is retrieved rather than recomputed.
priceable-unit-labels	A list of the priceable-unit-labels that the faring-atom enters into.

After the faring process **18** decomposes the itineraries into faring atoms, the faring process **18** retrieves fares **84** and rules **86** for each faring atom by accessing the fares/rules database **20***a* mentioned above. At this point a fare's routing is retrieved from a routing database and applied to a faring atom. If the routing test fails, the fare cannot be applied to the faring atom and a fare component is not built.

The faring process **18** applies the rules **88** to the faring atoms to produce fare components. Fare-components are combinations of faring-atoms and fares. Fare-components (TABLE 6) are produced if a fare's rules pass a preliminary check on a faring-atom. They are used to store deferred rules (e.g., deferred record-2s and combinability record-2s) that are applied at a later stage of processing **88***a*. Fare components also store extra information produced during the rule-checking process, such as information about surcharges and penalties and discounts that are applied to the base fare price.

TABLE 6

Fare-Component fields	Use
fare	The fare-component's fare.
faring-atom	The fare-component's faring-atom.
deferred-record-2s	A list of non-category-10 record-2s that have not been fully evaluated.
combinability-record-2	If the fare's rules include a category-10 ("Fare Combinability" record-2, it is stored here.
surcharges	A list of surcharges, penalties, discounts and other pieces of information produced during the rule-checking process.

From the fare components the faring process **18** constructs 90 priceable units. For certain types of rules such as those which require access to fares and/or flights from outside of the fare component, those rules are stored in the fare component for later or deferred evaluation. The priceable unit process **90**, takes valid fare components and constructs priceable units from the fare components. This process **90** involves grouping fare components from different slices and checking fare component combination restrictions. At this stage of processing, the rules deferred in step **88** are reapplied.

Priceable units are represented by priceable-unit-cores and priceable-unit-labels. Priceable-unit-cores are collections of fares and other information associated with fares within a priceable-unit, such as discounts and penalties and surcharges. Priceable-unit-cores (TABLE 7) are referenced by priceable-unit-labels.

TABLE 7

Priceable-Unit-Core fields	Use
fares	A list of fares.
slice-numbers	A list of the slices the fares originate from.
surcharges	A list of surcharges, penalties, discounts and other pieces of information produced during the rule-checking process.

Priceable-unit-labels group a set of priceable-unit-cores with sets of faring-atoms. Together, they are used to represent sets of priceable-units (TABLE 8).

TABLE 8

Proceable-Unit-Label fields	Use
priceable-unit-cores	A set of priceable-unit cores.
slice-numbers	A list of the slices the fares and faring-atoms originate from.
faring-atom-sets	A list of sets of faring-atoms, one per slice.

When all the fare components within a priceable unit are known, rules that were deferred from the processing **88** are applied **92** to the priceable unit sets of faring atoms.

After evaluation of the deferred record-2s at the priceable unit stage, the itineraries and priceable units are grouped together into complete set of pricing solutions. This occurs by a link process **94** that links itineraries to corresponding pricing units from different slices to provide **96** the pricing solution. At this juncture, any remaining cross priceable unit fare combinability checks are performed to eliminate invalid combinations.

The linking process involves two additional data structures slice-label-sets and open-label-sets. Slice-label-sets group itinerary divisions by the multi-slice priceable-unit-labels they can enter into. In each slice of a journey, a unique slice-label-set is constructed for every set of multi-slice priceable-unit-labels. Each slice-label-set stores both the set of multi-slice priceable-unit-labels and a set of itinerary-label-holders, which contain single-slice priceable-unit-labels on a per-itinerary basis. Each slice-label-set is a pair of an itinerary and a set of division-label-holders. Each of these division-label-holders is a pair of a division and a set of sets of single-slice priceable-unit-labels (TABLE 9).

TABLE 9

Slice-Label-Set fields	Use
multi-slice-PU-labels	A set of multi-slice PU-labels.
itinerary-label-holders	A set of itinerary-label-holders.
Itinerary-Label-Holder fields	**Use**
itinerary	An itinerary.
division-label-holders	A set of division-label-holders.
Division-Label-Holder fields	**Use**
division	An itinerary division.
single-slice-PU-label-sets	A set of sets of single-slice PU-labels.

Open-label-sets (TABLE 10) are used to summarize the state of the linking process **94**. Each is a set of "open" multi-slice priceable-unit-labels and a set of backward-links. Each of these backward-links is a pair of a slice-label-set and an open-label-set.

TABLE 10

Open-Label-Set fields	Use
open-PU-labels	A set of open multi-slice PU-labels.
backward-links	A set of backward-links.
Backward-Link fields	**Use**
slice-label-set	A slice-label-set.
open-label-set	An open-label-set.

The pricing solution resulting from the linking process **94** is used to construct a pricing graph from the various data structures built during the preceding processes. This pricing graph is transmitted to the client process or can be stored for later use or transmission. A pseudocode representation of the high level processing logic involved in the above search procedure is set out below in TABLE 11.

TABLE 11

```
price-itinerary-sets(itinerary-sets, fare-database, rule-database, routing-database, environmental-information)
    //
    // itinerary-sets is a set of sets of itineraries, one per slice.
    // environmental-information contains information about the passenger, the current date, the location
    // where tickets will be purchased, and other non-itinerary-based information that is necessary for applying
    // fare rules.
    //
    Let faring-market-sets = {}
    // Construct itinerary-divisions, faring-markets and faring-atoms.
    Let slice-number = 1
    For itinerary-set in itinerary-sets
        //
        // create-divisions constructs the itinerary-divisions, faring-markets and faring-atoms for
        // all the itineraries within a slice. It returns a set of faring-markets.
        faring-market-sets += create-divisions(itineraries, slice-number, fare-database)
        slice-number += 1
    // Apply fare rules, constructing fare-components in each faring-market.
    For faring-market-set in faring-market-sets
        //
        // apply-fare-rules constructs fare-components for each faring-market within a slice.
        // This process contains pseudo-code for apply-fare-rules.
        apply-fare-rules(faring-market-set, fare-database, rule-database,
                routing-database, environmental-information)
    // Create priceable-units.
    // for create-priceable-units
    create-priceable-units(faring-market-sets)
    // Link itineraries between slices. This procedure returns either nil, if there are no pricing-solutions, or
    // a "root" open-label-set. This process is described in link-itineraries
    Let root-object = link-itineraries(itinerary-sets)
    If (root-object = nil)
        return(nil)
    // Create the pricing-graph from the data-structures that have been built in the preceding steps.
    // This process includes psedo-code for create-pricing-graph.
    Let root-node = create-pricing-graph(root-object)
    // Return the pricing graph.
    return(root-node)
```

Referring now to FIG. **5**, the process **82** to decompose an itinerary into faring atoms includes a retrieval process **100** that retrieves all itineraries for all slices in a journey. For each itinerary in each slice, the process **82** groups faring atoms by faring markets at **104** and partitions itineraries into the divisions of faring atoms at **106**.

Referring now to FIG. **6**, itineraries are partitioned into divisions of faring atoms by examining **110** for each itinerary whether or not the itinerary includes more than one leg **112** on the same airline **114**. For each sequence on the same airline, a faring-atom is produced. If the sequence has more than one leg, the sequence is also split into multiple faring-atoms (at **116**), resulting in more than one division of the itinerary into a set of faring-atoms. The process checks **118** whether fares exist for the airline in the markets spanned by each faring atom. Otherwise, the process will branch from the examination process **112** and the airline check process **114** to a fare check process **118** to check in the fare database **20***a* that a fare exists for the airline in the market spanned by the faring atom. If all of the faring atoms within a division have at least one fare in the market, a division for the market is produced at **120**. Another possible implementation creates divisions by producing all possible partitions of legs into faring-atoms.

A high-level pseudocode representation for the algorithm that generates faring atoms, faring markets and faring divisions for each itinerary within a slice is set forth below in TABLE 12.

TABLE 12

```
create-divisions(itineraries, slice-number, fare-database)
    Let faring-atoms = {}
    Let faring-markets = {}
    Subroutine get-faring-market(origin-airport, destination-airport, airline)
        Let origin-city = city(origin-airport)
        Let destination-city = city(destination-airport)
        Let previous-faring-market = find(<origin-city, destination-city, airline>, faring-markets)
        If (previous-faring-market)
            return(previous-faring-market)
        Else
            If (fares-exist(origin-city, destination-city, airline))
                Let faring-market = new-faring-market( )
                faring-market.slice-number = slice-number
                faring-market.origiin-city = origin-city
                faring-market.destination-city = destination-city
                faring-market.airline = airline
                faring-market.faring-atoms = {}
                faring-markets += faring-market
                return(faring-market)
            Else
                return(nil)
    Subroutine get-faring-atom(legs-and-departure-times, origin-airport, destination-airport, airline)
        Let previous-faring-atom = find(legs-and-departure-times, faring-atoms)
        If (previous-faring-atom)
            return(previous-faring-atom)
        Else
            Let faring-market = get-faring-market(origin-airport, destination-airport, airline)
            If (faring-market <> nil)
                Let faring-atom = new-faring-atom( )
                faring-atom.faring-market = faring-market
                faring-atom.legs-and-departure-times = legs-and-departure-times
                faring-atom.priceable-unit-labels = {}
                faring-atom.cached-results = {}
                faring-market.faring-atoms += new-faring-atom
                faring-atoms += faring-atom
                return(faring-atom)
            Else
                return(nil)
    Subroutine get-online-divisions(legs-and-departure-times, origin-airport, destination-airport, airline)
        Let online-divisions = {}
        Let number-of-legs = length(legs-and-departure-times)
        Let single-faring-atom = get-faring-atom(legs-and-departure-times, origin-airport, destination-airport, airline)
        If (single-faring-atom <> nil)
            online-divisions += list(single-faring-atom)
        For i from 1 to number-of-legs - 1
            Let legs-and-departure-times1 = legs-and-departure-times[1 . . . i]
            Let legs-and-departure-times2 = legs-and-departure-times[i+1 . . . number-of-legs]
            Let destination-airport1 = destination-airport(faring-atom-legs1)
            Let origin-airport2 = origin-airport(faring-atom-legs2)
            If (is-not-same-flight-segment(legs-and-departure-times1, legs-and-departure-times2))
                Let faring-atom1 = get-faring-atom(legs-and-departure-times1, origin-airport,
                        destination-airport1, airline)
                Let faring-atom2 = get-faring-atom(legs-and-departure-times2, origin-airport2,
                        destination-airport, airline)
                If (faring-atom1 <> nil and faring-atom2 <> nil)
                    online-divisions += list(faring-atom1, faring-atom2)
        return(online-divisions)
    For each itinerary in itineraries
        Let divisions = { {} }
        Let legs-and-departure-times = itinerary.legs-and-departure-times
```

Referring now to FIG. **7**, a process **88** to apply the faring rules to faring atoms is shown. The input to the application process **88** includes the fare/rules database **20***a* and faring markets **130**. For each faring atom in each faring market, a fare and corresponding rules are retrieved **132** from fare/rules database **20***a*. The rules are applied to the faring-atoms at **134**. Because faring-atoms are shared across itineraries, it is only necessary to apply a fare's rules to a faring atom once no matter how many itineraries the faring-atom may appear in. The results of rule applications are cached **136**. Caching of a rule minimizes computational effort. This is because the rule application process **88** involves many redundancies, because different fares share rule restrictions. Valid fare/faring-atom combinations are stored **138** in the form of fare-components.

Referring to FIGS. **8**A and **8**B, a process **132** for retrieving rules and footnotes from the rules database **20***a* containing the ATPCO rules, routes and fares includes retrieving 150 general rules commonly referred to as record_**0**'s for each faring atom in a faring market. The retrieved general rules are searched **152** to find the first record_**0** matched to the faring atom to produce a matched record_**0**. If there is a matched record_**0**, it is stored at **154**. Whether or not there are matched record_**0**'s, the process **132** retrieves 156 application rules commonly referred to as record_**1** rules. The retrieved application rules are searched to find the first record_**1** matched to each of the faring atoms. The first matched record_**1**'s is stored **160**.

If after traversing through all the record_**1**'s there are no matches found, the process will return a "FAIL" at **162** and

terminate indicating that the faring atom cannot be used by the faring process **18**.

If there is a match, the process **132** retrieves 164 category controls (commonly referred to as record_2's). The process **132** will find the first record-2 in each category that matches the fare. Record_2's or the category control records typically comprise a large number of categories currently 30 different categories. The process is run for each category. It is not necessary that every category have a match and in many instances many if not most categories will not have a match. Rules in those categories that have a match are stored at **168** and the process continues to run at **170** until all categories have been traversed. If all categories have not been traversed, a pointer to a next category **172** is set and the next category is retrieved **164**. If all categories have been evaluated, the process retrieves 176 linked record_3's and applies process 132 (FIG. **7**). In this embodiment. Record-3's are retrieved **176** as part of the rule application process **132**.

The ATPCO rule retrieval process **132** that retrieves the rules for a fare includes record-matching processes **150, 156,** and **164** (FIG. **8A**) that may depend on the departure date of the faring-atom. To minimize computational effort expended in rule retrieval **132**, rules for a fare are not retrieved once for every faring-atom, but at most once per departure-date. To further minimize computation, a caching mechanism is employed.

Referring now to FIG. **8C**, a process that checks dates for rule retrieval includes retrieving **177** a current date from a faring market that contains faring atoms with multiple travel dates, and a stored date corresponding to a latest stored date that a result for the rule remains valid. The current date is compared **178** to the stored date and if the rule still remains valid (i.e., the current date falls within a bound set by the stored date) the rule is not retrieved and instead rules that had been cached are used. If the stored date for the rule is not valid then a new rule is retrieved **179** and a new date is subsequently stored **180** for the new rule.

Referring now to FIG. **9**, a process **134** for applying the rules retrieved with process **132** is shown. The rule application process **134** operates on each faring atom. The process **134** applies **181** the record-1 records to check for booking codes etc. The process **134** determines **182** whether each record-2 was cached in the faring atom. If a record-2 was cached in the faring atom, the process returns **183** the cached results. Otherwise, the process **134** applies **184** the record_2's for each of the stored record_2 categories. Rules provisions are expressed as "record-2s", which are retrieved **132**, as described in FIGS. **8A, 8B**. These record-2s express logical relations over so called "record-3s", which are records that contain detailed provisions. Individual procedures are provided for evaluating each record-3 as applied to a faring atom. Each record-3 procedure returns either DEFER, FAIL, PASS, NO-MATCH or UNAVAILABLE, and these results are combined according to the logic in the record-2 to produce a result of either DEFER, FAIL or PASS for the entire record-2. The proper procedures for applying record-3s and for combining their results according to record-2s are described in the ATPCO documentation. The "PASS" value is an extension used here since not all record-3s can be fully evaluated on the basis of the faring-atom alone. The RECORD-2 result is either PASS, FAIL or DEFER (the other two values are from record-3s).

As a result of returning a cached result or of the application of the record_2's, the process can return one of five possible results, "DEFER", "PASS", or "FAIL." The record as well as its results DEFER, PASS, or FAIL, are cached at **136** in the faring atom. The result FAIL causes the process **134** to exit **190**. Whereas, returning a pass or a defer permits the process **134** to continue to examine remaining record-2s. A defer or pass result is stored **185** and it is determined **186** whether all record-2s have been processed. If all records have not been applied/examined if cached (branches to NO), the next record-2 is retrieved at **186***a*. After all record-2s have been examined (YES at **186**) and, if pass results have been provided for all, e.g., did not branch, the PASS result causes the process **134** to construct **187** fare components and exit **190**. If at least one DEFER result was returned process **134** constructs **188** the fare components, stores **189** deferred record-2's in the faring component and exits **190**. The routines **188, 189** and **190** thus correspond to the stored valid faring atom combination routine **138** (FIG. **7**).

APPLICATION OF RECORD-3S

The information contained in record-3s varies by category. For example, category-5 record-3s, which specify advanced-purchase restrictions, contain fields that specify how long in advance of travel time a fare must be reserved, how long after a reservation a fare must be purchased, and so on. Category-4 record-3s, which specify flight-restrictions, contain fields that restrict flight-numbers, aircraft-type, airline, and whether flights are direct or non-stop. Every category has a different procedure for evaluating a faring-atom.

As discussed above the record-3 procedures that evaluate a faring-atom returns one of five values, and may return some other information such as discounts, penalties and surcharges. A value of PASS or FAIL can only be returned if that answer can be determined without examining any faring-atom other than the one the fare spans.

The ATPCO rules distinguish between fare-component and priceable-unit restrictions. Most restrictions on travel route, flight-numbers, and aircraft-type are fare-component-based, i.e., restrict only the flights in the faring-atom governed by the fare. On the other hand, minimum and maximum-stay restrictions are priceable-unit-based, i.e., apply to joint properties of all the faring-atoms within a priceable-unit. A minimum-stay requirement for a round-trip fare, for example, constrains the combination of outbound and return faring-atoms. Generally speaking, FC-based record-3s will be able to return either PASS or FAIL, while PU-based restrictions may need to be deferred. Deferring rules means checking them at a later point, however. This is a more computationally expensive process, because it must be done for combinations of faring-atoms within a priceable-unit, and the number ways faring-atoms can be combined to create priceable-units can be quite large, and grows quickly with the size of the priceable-unit. For this reason, whenever possible it is desirable for record-3 application not to result in a value of DEFER.

Many properties of faring-atoms can be bounded. For example, the earliest and latest departure-time within a faring-market, or within a slice, can be recorded, as well as the minimum and maximum number of connections within the faring-market and so forth. This information can often be used to evaluate priceable-unit restrictions at the fare-component level. A simple example of this is given below.

In this example, it is assumed that a certain fare's rules require at least a 3-day layover at the intermediate point of a round-trip priceable-unit, measured from the departure-times of the fare-components. The fare is used for the first half (outbound travel) of the priceable-unit, in the NW CHI_MSP faring-market in slice 1. If there are exactly two slices in the query, then the fare-component that covers return travel must come from the NW MSP_CHI faring-

market in slice 2. Suppose that the following faring-atoms exist (TABLE 13). (The airport ORD is in the city Chicago.)

TABLE 13

Slice 1 NW CHI_MSP faring-atoms	Slice 2 NW MSP_CHI faring-atoms
ORD_MSP NW220 12APR97 13:00	MSP_ORD NW301 15APR97 19:00
ORD_MSP NW220 13APR97 13:00	MSP_ORD NW577 16APR97 12:00
ORD_MSP NW220 14AFR97 13:00	MSP_ORD NW301 16APR97 19:00

In each faring-market, the earliest and latest departure-times can be calculated. In this case, the earliest departure-time in the slice-2 NW MSP_ORD market is 15APR97 19:00, and the latest departure-time is 16APR97 19:00.

When the minimum-stay requirement restriction is applied to the first faring-atom, its departure time of 12APR97 13:00 can be compared to the two outer bounds on return-travel departure-time, 15APR97 19:00 and 16APR97 19:00. In this case, the minimum-stay requirement is met even for the earliest possible return travel time, so the faring-atom unconditionally passes the restriction.

Similarly, for the third faring-atom, since the restriction fails even for the latest possible return-travel departure-time, the faring-atom unconditionally fails the minimum-stay requirement. But for the second faring-atom, because the restriction fails for the earliest possible return time, but passes the latest possible return time, it is necessary to defer the application of the restriction.

GENERAL TIME BOUNDS

Many priceable-unit-based categories restrict times. Categories 3, 5, 6, 7, 8, 9, 10 and 14 are usually priceable-unit-based. Categories 3 and 14 usually restrict the departure-date of the first flight in a priceable-unit. Category 5 specifies how far in advance of travel fares must be purchased, and this is usually measured from the departure-date of the first flight in a priceable-unit. Categories 6 and 7 specify minimum and maximum-stays at stopovers within a priceable-unit.

In many cases these categories do not need to be deferred. This is especially true if, as in the above example, time-bounds are known for other faring-markets in the journey, and the range of faring-markets that might enter into a priceable-unit with the faring-atom in not great. It is a relatively simple matter to record for each faring-market the earliest and latest departure-date of any faring-atom within the faring-market. This can be done as faring-atoms are constructed. The problem remains of how to know what other faring-markets might participate in a priceable-unit with the faring-atom at hand.

CATEGORY-3

Pseudo code for an example of a procedure that implements record-3 category-3, "Seasonality Restrictions" is shown in TABLE 15. Each category-3 record-3 an example of which is shown in TABLE 14 specifies a permissible date range for travel, via a start-date and an end-date, either of which may be left blank. The default interpretation of category-3 is that these date restrictions apply to the departure-date of the first flight of the priceable-unit. This interpretation can be modified in two ways. First, if a certain field is set, then the category becomes fare-component based. In other words, the date restrictions apply to the departure-date of the first flight within the fare-component. Second, a geographic specification may be provided that alters the measurement of the departure-date. For example, the geographic specification may dictate that the relevant date is the departure-date of the first transoceanic flight.

Category-3s (TABLE 14) also includes a field that specifies whether the record-3 is available. If it is not, that is an indication that some information is missing and the record-3 should not be used for pricing a ticket. In this case, the record-3 application must return UNAVAILABLE. Finally, a category-3 may include a specification of a date range that the category-3 is valid for. If travel is outside of these dates, the record-3 application must return NO-MATCH.

TABLE 14

Category-3 field	Example
Earliest Permitted Travel Date	nil
Latest Permitted Travel Date	190CT97
Fare-Component Based	false
Geographic Specification	nil
Earliest Record-3 Valid Date	15MAY88
Latest Record-3 Valid Date	nil
Available	true

The logic of the procedure that processes the record-3 is as follows. If the record-3 is not available, UNAVAILABLE is returned. If travel is outside of the valid date-range of the record-3, NO-MATCH is returned. Then, processing branches depending on whether the record-3 is priceable-unit based (the default), or fare-component based. If fare-component based, and there is no geographic specification, the departure date of the faring-atom is compared to the date-range of the record-3, and either PASS or FAIL is returned. If a geographic specification is provided, then this is used to compute the relevant travel date, and the same procedure applies. If, on the other hand, the record-3 is priceable-unit based, then broad time-bounds checks are used. If there is no geographic specification, the earliest and latest possible priceable-unit departure-dates are retrieved and compared to the date-range of the record-3. If they both succeed, PASS is returned. If they both fail, FAIL is returned. Otherwise DEFER is returned. Finally, if the record-3 is priceable-unit based and includes a geographic specification, then DEFER is returned. The following pseudo-code implements the processing of record-3 category-3 in the case where the record-3 must be evaluated given only a single faring-atom from the priceable-unit.

TABLE 15

```
apply-record-3-FC-category-3(record-3, faring-atom, passenger-information, current-date)
    If (record-3.available = false)
        return(unavailable)
    Let date = departure-date(faring-atom)
    If ((record-3.earliest-record-3-valid-date <> nil and record-3.earliest-record-3-valid-date > date) or
            (record-3.latest-record-3-valid-date <> nil and record-3.latest-record-3-valid-date < date))
        return(no-match))
    If (record-3.fare-component-based = true)
```

TABLE 15-continued

```
    Let travel-date = date
    If (record-3.geographic-specification <> nil)
        travel-date = apply-geographic-specification(record-3.geographic-specification, faring-atom)
    If ((record-3.earliest-permitted-travel-date <> nil and record-3.earliest-permitted-travel-date > travel-date) or
            (record-3.latest-permitted-travel-date <> nil and record-3.latest-permited-travel-date < travel-date))
        return(fail)
    Else
        return(pass)
Else If (record-3.geographic-specification <> nil)
    return(defer)
Else
    Let earliest-travel-date = earliest-priceable-unit-departure-date(faring-atom)
    Let latest-travel-date = latest-priceable-unit-departure-date(faring-atom)
    Let result = pass
    If (record-3.earliest-permitted-travel-date <> nil)
        If (record-3.earliest-permitted-travel-date > latest-travel-date)
            result = fail
        Else If (record-3.earliest-permitted-travel-date <= earliest-travel-date)
            result = defer
    If (record-3.latest-permitted-travel-date <> nil)
        If (record-3.latest-permitted-travel-date < earliest-travel-date)
            result = fail
        Else If (result <> fail and record-3.latest-permitted-travel-date >= latest-travel-date)
            result = defer
    return(result)
```

There can be another version of this application procedure, as shown in TABLE 16 dedicated to the case where all of the faring-atoms within the priceable-unit are known. This procedure is simpler, because there is no need for time bound checks since all times are known exactly. This procedure is used to evaluate deferred record_3's (see TABLE 24).

TABLE 16

```
apply-record-3-PU-category-3(record-3, fares, faring-atoms, prime-faring-atom, passenger-information, current-date)
    If (record-3.available = false)
        return(unavailable)
    Let date = departure-date(prime-faring-atom)
    If ((record-3.earliest-record-3-valid-date <> nil and record-3.earliest-record-3-valid-date > date) or
            (record-3.latest-record-3-valid-date <> nil and record-3.latest-record-3-valid-date < date))
        return(no-match))
    Let travel-date = date
    If (record-3.fare-component-based = true)
        If (record-3.geographic-specification <> nil)
            travel-date = apply-geographic-specification(record-3.geographic-specification, prime-faring-atom)
    Else
        travel-date = departure-date(faring-atoms)
        If (record-3.geographic-specification <> nil)
            travel-date = apply-geographic-specification(record-3.geographic-specification, faring-atoms)
    If ((record-3.earliest-permitted-travel-date <> nil and record-3.earliest-permitted-travel-date > travel-date) or
        (record-3.latest-permitted-travel-date <> nil and record-3.latest-permited-travel-date < travel-date))
        return(fail)
    Else
        return(pass)
```

Referring now to FIG. **10**, the process **90** for constructing priceable units is shown. The term "priceable unit" as used herein represents a fundamental unit at which many fare restrictions apply. For example, round trip fares often include minimum stay requirements and these can only be expressed when both an outbound and a return faring atom are combined. This occurs at the level of the priceable unit.

The process **90** of constructing priceable unit cores and pricing unit labels is organized as several nested procedures as follows. The process enumerates **200** a collection of faring markets. Collections of faring markets are enumerated **200** with each faring market from a different slice by an algorithm that depends on the type of a priceable unit that is constructed. For example, for a round trip priceable unit two faring markets are chosen on the same carrier and between the same cities but in opposite directions. The process **90** also enumerates collections of sets of faring components at **202**. For each faring market in a collection of faring markets its faring components are partitioned into sets of fare components that have the same fare and the same combinability record-2s. Collections of these sets are enumerated with one set chosen from each faring market resulting in a collection of fares and associated collection of sets of fare components. At this juncture, any combinability record 2-s are evaluated to insure that the fares may be used together in a priceable unit.

The process **90** also enumerates **204** collections of fare components. Thus, given a collection of sets of fare components from **202**, the process evaluates any deferred record 2-s on collections of fare components in the enumeration process **204**. These collections are constructed by selecting one fare component from each set. The process of evaluating deferred rules on collections of fare components outputs results in the form of a list of factored representations of priceable units. Thus, the output is a logical OR of logical ANDs of logical ORs of fare components.

From the factored representations produced in **204** the process produces priceable-unit-labels **206** and priceable-unit-cores **208** with some sharing occurring between these structures to ensure that the number of priceable-unit-cores and priceable-unit-labels is kept to a minimum.

The pseudo-code below (TABLE 17) summarizes enumerating collections of faring markets process **200**. It takes as input a set of sets of faring-markets, containing one set per slice. These are the faring-markets generated by the calls to "create-divisions," (**120** FIG. **6**), as described above.

TABLE 17

```
create-priceable-units(faring-market-sets)
    Let number-of-slices = length(faring-market-sets)
    Let priceable-unit-labels = {}
    For slice-number1 from 1 to number-of-slices
    For faring-market1 in faring-market-sets[slice-number1]
        Let airline = faring-market1.airline
        // Create one-way priceable-units.
        priceable-unit-labels = create-PUs-in-markets1 (faring-market1, priceable-unit-labels, one-way)
        For slice-number2 from slice-number1 + 1 to number-of-slices
        For faring-market2 in faring-markets-on-carrier(faring-market-sets[slice-number2], airline)
            // Create single and double open-jaws.
            priceable-unit-labels = create-PUs-in-markets2(faring-market1, faring-market2,
                                        priceable-unit-labels, open-jaw)
            If (faring-market1.destination-city = faring-market2.origin-city)
                // Create round-trips and circle-trips of size 2.
                If (faring-market2.destination = faring-market1.origin)
                    priceable-unit-labels = create-PUs-in-markets2(faring-market1, faring-market2,
                                            priceable-unit-labels, round-trip)
                    priceable-unit-labels = create-PUs-in-markets2(faring-market1, faring-market2,
                                            priceable-unit-labels, circle-trip2)
                For slice-number3 from slice-number2 + 1 to number-of-slices
                For faring-market3 in faring-markets-on-carrier(faring-market-sets[slice-number3], airline)
                    If (faring-market2.destination-city = faring-market3.origin-city)
                        // Create circle-trips of size 3.
                        If (faring-market3.destination-city = faring-market1.origin-city)
                            priceable-unit-labels =
                                create-PUs-in-markets3(faring-market1, faring-market2, faring-
                        //
                        // More iterations for circle-trips of lengths 4 and 5.
                        //
                        . . .
    // Store priceable-unit-labels in faring-atoms.
    For priceable-unit-label in priceable-unit-labels
        For faring-atom-set in priceable-unit-label.faring-atom-sets
            For faring-atom in faring-atom-set
                faring-atom.priceable-unit-labels += priceable-unit-label
```

This pseudo-code iterates over faring-markets in different slices, and passes faring markets to one of several possible create-PUs-in-markets procedures. These procedures vary by size of priceable-unit produced. The code ensures that the faring-markets are in the correct format for the type of priceable-unit produced, and that the priceable units are all on the same airline. This last restriction is motivated by efficiency since rarely do carriers permit priceable-units with fares from more than one airline.

Each call to create-PUs-in-markets returns an updated set of priceable-unit-labels. At the end of the procedure, these priceable-unit-labels are stored in their component faring-atoms.

There are many other combinability restrictions that limit the manner in which fare components can be combined into priceable continuous units. Even when searching for fares for a small number of itineraries, there can be a very large number of possible pricing units because of the large number of possible fares that can exist. It is preferred to represent these priceable units in a compact manner so as to minimize the computation involved in their construction.

The faring algorithm does not actually construct a data-structure for every priceable-unit. Instead, priceable-units are represented by a combination of two data structures: priceable-unit-cores (PU-cores) and priceable-unit-labels (PU-labels). PU-core data structures contain all the information associated with an individual priceable-unit except its faring-atoms. Thus, each PU-core contains a set of fares (one fare per fare-component in the priceable-unit) and any other information associated with those fares, such as discounts, surcharges and penalties. PU-label data structures compactly represent connections between faring-atoms and PU-cores.

At this stage of processing, a collection of fares has been fixed on, and for each fare there is a set of fare-components. Priceable-units are constructed by selecting one fare-component from each set and evaluating any deferred rules. The simplest manner that this could be accomplished would be to enumerate complete collections of fare-components and to apply the deferred record-2s from within these fare-components. Often, this method can be made more efficient in some cases by use of the function get-OR-AND-OR-form as will be described. That function takes a collection of sets of fare-components, evaluates any deferred rule-conditions, and returns a representation of the set of valid priceable-units. This representation is in OR-AND-OR form. In other words, it takes the form of a set of collections of sets of fare-components. This is very close to a set of priceable-unit-labels except that since the sets are of fare-components rather than faring-atoms, there are no PU-cores. The inner sets of fare-components returned by get-OR-AND-OR-form are guaranteed to have the same fares, surcharges, discounts, penalties and so on.

PU-cores and PU-labels are constructed from the output of get-OR-AND-OR. The pseudo-code below summarizes this procedure. It iterates over the inner AND-OR form, constructing PU-cores (if no identical PU-core already exists) and PU-labels (if no identical PU-label already exists). PU-labels are constructed by mapping from fare-components to faring-atoms. PU-cores are stored on PU-labels.

TABLE 18

```
create-PUs-from-fare-components(faring-markets, fares, fare-component-sets, existing-PU-labels,
                                environmental-information)
    Let slice-numbers = {}
    Let PU-labels = existing-PU-labels
    Let PU-cores = {}
    For faring-market in faring-markets
        slice-numbers += faring-market.slice-number
    For AND-OR in get-OR-AND-OR(faring-markets, fare-component-sets, environmental-information)
        //
        // AND-OR is a collection of sets of fare-components, representing all the priceable-units
        // that can be constructed by choosing one fare-component from each set.
        //
        Let PU-core = nil
        Let surcharges = {}
        Let faring-atom-sets = {}
        For fare-component-set in AND-OR
            surcharges += first(fare-component-set).surcharges
            Let faring-atom-set = {}
            For fare-component in fare-component-set
                faring-atom-set += fare-component.faring-atom
            faring-atom-sets += faring-atom-set
        // Find an existing PU-core with these fares and surcharges, or construct a new one.
        For test-PU-core in PU-cores
            If (test-PU-core.surcharges = surcharges)
                PU-core = test-PU-core
        If (PU-core = nil)
            PU-core = new-PU-core( )
            PU-core.fares = fares
            PU-core.surcharges = surcharges
            PU-core.slice-numbers = slice-numbers
        // Find an existing PU-label with these faring-atoms, or construct a new one.
        Let PU-label = nil
        For test-PU-label in PU-labels
            If (test-PU-label.faring-atom-sets = faring-atom-sets)
                PU-label = test-PU-label
        If (PU-label = nil)
            PU-label = new-PU-label( )
            PU-label.faring-atom-sets = faring-atom-sets
            PU-label.slice-numbers = slice-numbers
            PU-label.priceable-unit-cores = {}
            PU-labels += PU-label
        PU-label.priceable-unit-cores += PU-core
    return(PU-labels)
```

To understand the role PU-cores and PU-labels play in the faring algorithm, it may be helpful to look at an example, involving a round-trip journey between BOS and MSP. In this example, there are four outbound itineraries and four return itineraries, each of which is spanned by a single faring-market.

For both the outbound and return itineraries, there is a choice between two airlines, UA and NW. Both of these airlines offer two round-trip fares and one one-way fare. This situation is summarized in TABLE 19 below.

TABLE 19

Faring-market	Faring-Atoms	Fares
Slice 1: UA BOS→MSP	BOS→MSP UA100	UA BOS-MSP RT "Q"
	BOS→MSP UA200	UA BOS-MSP RT "M14"
		UA BOS-MSP OW "Y"
Slice 1: NW BOS→MSP	BOS→MSP NW300	NW BOS-MSP RT "HE7"
	BOS→MSP NW400	NW BOS-MSP RT "Q7NR"
		NW BOS-MSP OW "F"
Slice 2:	MSP→BOS UA111	same as for the outbound

TABLE 19-continued

Faring-market	Faring-Atoms	Fares
UA MSP→BOS	MSP→BOS UA222	UA faring-market
Slice 2:	MSP→BOS NW333	same as for the outbound
NW MSP→BOS	MSP→BOS NW444	NW faring-market

Assume that in each of the four faring-markets (i.e., BOS-MSP UA, BOS-MSP NW, MSP-BOS UA and MSP-BOS NW), fare-components have been constructed for every combination of faring-atom and fare. The fare-components built from the fare "NW BOS-MSP RT HE7" contain a deferred record-2 that is checked during priceable-unit construction. This record-2 does not permit outbound travel on flight "NW300" combined with return travel on flight "NW444." When constructing round-trip priceable-units, round-trip fares are combined with like round-trip fares. This situation permits the construction of a total of 23 priceable-units, as shown in TABLE 20.

TABLE 20

Priceable-Unit Number and Type	Slice-1 Faring-Atom	Slice-1 Fare	Slice-2 Faring-Atom	Slice-2 Fare
1 Round-trip	BOS→MSP UA100	RT "Q"	MSP→BOS UA111	RT "Q"
2 Round-trip	BOS→MSP UA100	RT "M14"	MSP→BOS UA111	RT "M14"
3 Round-trip	BOS→MSP UA100	RT "Q"	MSP→BOS UA222	RT "Q"
4 Round-trip	BOS→MSP UA100	RT "M14"	MSP→BOS UA222	RT "M14"
5 Round-trip	BOS→MSP UA200	RT "Q"	MSP-BOS UA111	RT "Q"
6 Round-trip	BOS→MSP UA200	RT "M14"	MSP→BOS UA111	RT "M14"
7 Round-trip	BOS→MSP UA200	RT "Q"	MSP→BOS UA222	RT "Q"
8 Round-trip	BOS→MSP UA200	RT "M14"	MSP→BOS UA222	RT "M14"
9 Round-trip	BOS→MSP NW300	RT "HE7NR"	MSP→BOS NW333	RT "HE7"
10 Round-trip	BOS→MSP NW300	RT "Q7NR"	MSP→BOS NW333	RT "Q7NR"
11 Round-trip	BOS→MSP NW300	RT "Q7NR"	MSP→BOS NW444	RT "Q7NR"
12 Round-trip	BOS→MSP NW400	RT "HE7NR"	MSP→BOS NW333	RT "HE7"
13 Round-trip	BOS→MSP NW400	RT "Q7NR"	MSP→BOS NW333	RT "Q7NR"
14 Round-trip	BOS→MSP NW400	RT "HE7NR"	MSP→BOS NW444	RT "HE7"
15 One-way	BOS→MSP NW400	RT "Q7NR"	MSP→BOS NW444	RT "Q7NR"
16 One-way	BOS→MSP UA100	OW "Y"		
17 One-way	BOS→MSP UA200	OW "Y"		
18 One-way	BOS→MSP NW300	OW "F"		
19 One-way	BOS→MSP NW400	OW "F"		
20 One-way			MSP→BOS UA111	OW "Y"
21 One-way			MSP→BOS UA222	OW "Y"
22 One-way			MSP→BOS NW333	OW "F"
23 One-way			MSP→BOS NW444	OW "F"

Even in this example, the list of possible priceable-units is long (23 units). The reason that there are so many priceable-units is because production of priceable-units involves several choices (of fares and faring-atoms).

In TABLE 21 below, each entry represents a choice (an OR) of either faring-atoms or PU-cores. Each row represents a collection (an AND) of these choices. And finally, the entire table represents a choice (an OR) over these collections. Collectively, this OR-AND-OR table provides a compact representation of the 23 priceable-units.

TABLE 21

Label Number	Slice-1 Faring-Atom Options	Slice-2 Faring-Atom Options	Priceable-Unit-Core Options
1	BOS→MSP UA100	MSP→BOS UA111	1: RT "Q", 2: RT "Q"
	BOS→MSP UA200	MSP→BOS UA222	1: RT "M14", 2: RT "M14"
2	BOS→MSP NW300	MSP→BOS NW333	1: RT "Q7NR", 2: RT "Q7NR"
	BOS→MSP NW400	MSP→BOS NW444	
3	BOS→MSP NW300	MSP→BOS NW333	1: RT "HE7", 2: RT "HE7"
4	BOS→MSP NW400	MSP→BOS NW333	1: RT "HE7", 2: RT "HE7"
		MSP→BOS NW444	
5	BOS→MSP UA100		1: OW "Y"
	BOS→MSP UA200		
6	BOS→MSP NW300		1: OW "F"
	BOS→MSP NW400		
7		MSP→BOS UA111	2: OW "Y"
		MSP→BOS UA222	
8		MSP→BOS NW333	2: OW "F"
		MSP→BOS NW444	

Each row of TABLE 21 is a priceable-unit-label (PU-label), an object that factors a set of priceable-units into a collection of choices that have no further dependencies. There is a choice for every faring-atom involved in the priceable-unit, and a choice of a priceable-unit-core (PU-core). Each PU-core contains the same number of fares as there are faring-atom choices. In the case where there are no constraints between faring-atoms in different slices, PU-labels are a very compact representation of priceable-units. PU-label #1, for example, represents a total of eight different priceable-units. In cases where there are interactions between the faring-atoms in different slices, several PU-labels can be produced for a single PU-core. An example of several PU-labels is shown for the NW RT "HE7" fare represented by PU-labels numbers 3 and 4. These priceable-unit-labels and priceable-unit-cores are used by the linking procedure **94** (FIG. 4B) to associate itineraries from more than one slice to fares.

ENUMERATING COLLECTIONS OF SETS OF FARE-COMPONENTS

Each of the faring-markets that is passed to create-PUs-in-markets has a set of fare-components produced by applying fare rules procedure **88**.

Referring now to FIG. **10A**, enumerating collections of sets of fare-components **202** (described by pseudo-code below) partitions the fare-components in each faring-market into sets such that every fare-component in a set has the same fare and the same combinability record-2s. Fare com-

binability restrictions are specified in record-2s rule category 10. Any category-10 record-2s in a fare's rules is stored in the combinability-record-2 field in the fare-components.

Once fare-components are partitioned **202***a* into sets, collections of these sets are enumerated **202***b*, by selecting one set from each faring-market. For each fare there is a choice of fare-components. At this point, when the fares within a priceable-unit have been fixed, any category-10 record-2s that was deferred is applied **202***c* to determine whether the fares may be used together in a priceable-unit. This is accomplished by applying **202***c* each fare's combinability record-2 (if it has one) to every other fare within the priceable-unit.

The code below (TABLE 22) is written for two faring-markets within a priceable-unit, as would be the case for round-trips, open jaws and circle-trips of size two. Similar code would be used for priceable-units of other sizes.

TABLE 22

```
partition-fare-components-into-sets(faring-market)
    //
    // Partition fare-components into sets based on fare and combinability record-2.
    //
    Let fare-component-sets = {}
    For fare-component in faring-market.fare-components
        Let fare = fare-component.fare
        Let combinability-record-2 = fare-component.combinability-record-2
        Let previous-set = nil
        For test-set in fare-component-sets
            Let test-fare-component = first(test-set)
            If ((fare = test-fare-component.fare) and
                (combinability-record-2 = test-fare-component.combinability-record-2))
                previsous-set = test-set
        If (previous-set = nil)
            fare-component-sets += list(fare-component)
        Else
            previous-set += fare-component
    return(fare-component-sets)
create-PUs-in-markets2(faring-market1, faring-market2, existing-PU-labels, PU-type, environmental-information)
    Let fare-component-sets1 = partition-fare-components-into-sets(faring-market1)
    Let fare-component-sets2 = partition-fare-components-into-sets(faring-market2.)
    Let PU-labels = existing-PU-labels
    For fare-components1 in fare-component-sets1
        For fare-components2 in fare-component-sets2
            Let fare1 = first(fare-components1).fare
            Let fare2 = first(fare-components2).fare
            Let combinability-record-2-1 = first(fare-components1).combinability-record-2
            Let combinability-record-2-2 = first(fare-components2).combinability-record-2
            // Check fare combinability requirements, by applying each fares's combinability
            // record-2 to all the other fares within the priceable-unit.
            If ((combinability-record-2-1 = nil or
                apply-combinability-record-2(combinability-record-2-1, fare2, PU-type) = pass) and
                (combinability-record-2-2 = nil or
                apply-combinability-record-2(combinability-record-2-2, fare1, PU-type) = pass))
                PU-labels = create-PUs-from-fare-components(list(faring-market1, faring-market2).
                                                            list(fare1, fare2),
                                                            list(fare-components1, fare-components2),
                                                            PU-labels, environmental-information)
    return(PU-labels)
```

The procedure create-PUs-in-markets2, after it has selected two fares and two sets of fare-components and verified fare combinability restrictions, calls the procedure **202***d* create-PUs-from-fare-components to evaluate deferred rules and construct priceable-unit-labels and priceable-unit-cores.

CONSTRUCTING PRICEABLE-UNITS

At this stage of processing, a collection of fares is determined, and for each fare there is a set of fare-components. Priceable-units are constructed by selecting one fare-component from each set and evaluating any deferred rules.

Referring now to FIG. **11**, a process **204** to enumerate collection of fare components is shown. The process **204** can enumerate **212** a collection of sets of fare-components, enumerate **214** fare components by selecting one component from each set, apply or evaluate **216** any deferred rule-conditions, and return a compact representation **218** of the set of valid priceable-units. A preferred technique to accomplish this uses a "GET_OR_AND_OR" operation described below TABLES 24 , 26 and 27.

The representation process **218** produces an OR-AND-OR representation of the priceable units. The process **218** produces a set of collections of sets of fare-components similar to that in FIG. **20**, that will later be transformed into priceable-unit-labels and priceable-unit-cores by processes **206** and **208** described further in TABLE 22. The inner sets of fare-components returned by get-OR-AND-OR-form have the same fares, surcharges, discounts, penalties and so on.

The procedure **218** get-OR-AND-OR, takes a collection of fare-component sets and enumerate collections of fare-components by selecting one from each set. It evaluates any deferred record-2s, and constructs a set of valid priceable-units. This set is transformed into a factored OR-AND-OR form, and returned.

Referring back to FIG. **10**, PU-cores and PU-labels are constructed **210** at process **206** and **208** from the output of get-OR-AND-OR process **204**. The pseudo-code below and FIGS. **12–13** summarizes this procedure. Construction **210** iterates over the inner AND-OR form, producing PU-cores **206** (if no identical PU-core already exists) and PU-labels **208** (if no identical PU-label already exists). PU-labels are

produced by mapping fare-components to faring-atoms and PU-cores are stored on PU-labels.

FACTORING PRICEABLE-UNITS

Referring now to FIGS. **12–15**, the get-OR-AND-OR process **218** to construct the priceable unit representation is shown and is also described in detail below through pseudo-code. As shown in FIG. **12** and in pseudo-code in TABLE 23 below, three different high-level procedures are used, depending on whether priceable-units are one **220**, two **222**, or three or more **224** fare-components.

TABLE 23

```
get-OR-AND-OR(faring-markets, fare-component-sets, environmental-information)
    Let size = length(faring-markets)
    If (size = 1)
        return(get-OR-AND-OR-1(faring-markets, fare-component-sets, environmental-information)
    Else If (size = 2)
        return(get-OR-AND-OR-2(faring-markets, fare-component-sets, environmental-information)
    Else
        return(get-OR-AND-OR-3+(faring-markets, fare-component-sets, environmental-information)
```

Two auxiliary functions are used in the get-OR-AND-OR procedures. The first, apply-deferred-record-2s **218***a*, takes a collection of fare-components (a potential priceable-unit) and evaluates any deferred record-2s they might have. In contrast to the initial stage of rule application, here all the fares and faring-atoms within the priceable-unit are known. It returns either PASS, FAIL or DEFER (in this case, DEFER means that the record-2s cannot be evaluated even at the priceable-unit level: they involve journey-level constraints). In such a case the priceable-unit is not produced (TABLE 24.)

TABLE 24

```
apply-deferred-record-2s(fare-components, environmental-information)
    Let faring-atoms = {}
    Let fares = {}
    For fare-component in fare-components
        fares += fare-component.fare
        faring-atoms += fare-component.faring-atom
    For fare-component in fare-components
        For record-2 in fare-component.deferred-record-2s
            If (apply-record-2-PU(record-2, fares, faring-atoms,
                                  fare-component.faring-atom,
                                  environmental-information)
                <> pass
                return(fail)
    return(pass)
```

The second auxiliary function, partition-fare-components-by-surcharges **218***b*, (TABLE 25) takes a set of fare-components and partitions it into subsets that have the same secondary characteristics: the same surcharges, discounts, penalties, etc.

TABLE 25

FACTORING PRICEABLE-UNITS OF SIZE ONE

```
partition-fare-components-by-surcharges(fare-components)
Let partitions = {}
For fare-component in fare-components
    Let found-partition = false
    Let surcharges = fare-component.surcharges
```

TABLE 25-continued

FACTORING PRICEABLE-UNITS OF SIZE ONE

```
    For partition in partitions
        If (first(partition).surcharges = surcharges)
            found-partition = true
            partition += fare-component
    If (found-partition = false)
        partitions += list(fare-component)
return(partitions)
```

Referring now to FIG. **13**, the get-OR-AND-OR function **230** for priceable-units of one fare-component (one-way priceable-units) is shown. It is passed only a single fare-component set **232**, and applies **216** deferred record-2s. The final set of valid fare-components is partitioned **234** into sets with the same surcharges, penalties, and discounts. The partitioned sets **236** are placed into the proper OR-AND-OR representation **238** to represent them in a compact form.

The procedure PU-1 for a priceable unit of size 1 is set out in TABLE 26.

TABLE 26

```
get-OR-AND-OR-1(faring-markets, fare-component-sets, environmental-information)
    Let valid-fare-components = {}
    For fare-component in first(fare-component-sets)
        If (apply-deferred-record-2s(list(fare-component), environmental-information)
            valid-fare-components += fare-component
    Let OR-AND-OR = {}
    For OR in partition-fare-components-by-surcharges(valid-fare-components)
        Let AND-OR = list(OR)
        OR-AND-OR += AND-OR
    return(OR-AND-OR)
```

FACTORING PRICEABLE-UNITS OF SIZE TWO

Referring to FIG. **14**, two-component priceable-units include round-trips and two-component open-jaws and circle-trips. They are common and should be computed efficiently and represented compactly. The function get-OR-AND-OR-2 **240** efficiently computes and represents two component priceable units (actions **242–246**). Combinations of fare-components are not enumerated unless it is necessary for evaluation of deferred record-2s. The resulting set of priceable-units is also represented in a compact OR-AND-OR form.

Pseudo code for the get-OR-AND-OR-2 procedure is set forth below (TABLE 27). The process **240** enumerates priceable units **248** by selecting one fare component from each set. The get-OR-AND-OR-2 process **240** tests deferred record-2s and, if the test is passed, adds the resulting valid priceable unit to the OR-AND-OR representation.

TABLE 27

```
get-OR-AND-OR-2(faring-markets, fare-component-sets, environmental-information)
    Let OR-AND-OR = {}
    Subroutine find-AND-OR(surcharges1, fare-component-set2)
        //
        // Return any AND-OR from OR-AND-OR that has fare-components with surcharges surcharges1 in its
        // first set of fare-components, and has fare-component-set2 as its second set of fare-components.
        //
        For AND-OR in OR-AND-OR
            If (first(first(AND-OR)).surcharges = surcharges1 and second(AND-OR) = fare-component-set2)
                return(AND-OR)
        return(nil)
    Subroutine add-uniform-cross-product(fare-component-set1, fare-component-set2)
        //
        // Add the priceable-units that can be had by selecting one element from fare-component-set1 and
        // one element from fare-component-set2. Both sets are uniform with respect to surcharges.
        //
        Let AND-OR = find-AND-OR(first(fare-component-set1).surcharges, fare-component-set2)
        If (AND-OR = nil)
            OR-AND-OR += list(fare-component-set1, fare-component-set2)
        Else
            first(AND-OR) = append(first(AND-OR), fare-component-set1)
    Subroutine add-cross-product(fare-component-set1, fare-component-set2)
        //
        // Add the priceable-units that can be had by selecting one element from fare-component-set1 and
        // one element from fare-component-set2.
        //
        Let uniform-fare-component-sets1 = partition-fare-components-by-surcharges(fare-component-set1)
        Let uniform-fare-component-sets2 = partition-fare-components-by-surcharges(fare-component-set2)
        For uniform-fare-component-set1 in uniform-fare-component-sets1
            For uniform-fare-component-set2 in uniform-fare-component-sets2
                add-uniform-cross-product(uniform-fare-component-set1, uniform-fare-component-set2)
    Subroutine enumerate-priceable-units(fare-component-set1, fare-component-set2)
        //
        // Enumerate priceable-units by selecting one fare-component from each set. Test deferred record-2s,
        // and if they pass, add the resulting priceable-unit to the OR-AND-OR representation.
        //
        For fare-component1 in fare-component-set1
            Let valid-fare-components2 = {}
            For fare-component2 in fare-component-set2
                If (apply-deferred-record-2s(list(fare-component1, fare-component2), environmental-information))
                    valid-fare-components2 += fare-component2
            If (valid-fare-components2 <> {})
                add-cross-product(list(fare-component1), valid-fare-components2)
    Let fare-component-set1-with-rules = {}
    Let fare-component-set1-without-rules = {}
    Let tare-component-set2-with-rules = {}
    Let fare-component-set2-without-rules = {}
For fare-component1 in first (fare-component-sets)
        If (fare-component1.deferred-record-2s = nil)
            fare-component-set1-without-rules += fare-component1
        Else
            fare-component-set1-with-rules += fare-component1
    For fare-component2 in second(tare-component-sets)
        If (fare-component2.deferred-record-2s = nil)
            fare-component-set2-without-rules += fare-component2
        Else
            fare-component-set2-with-rules += fare-component2
    // There is no need to enumerate combinations of fare-components that have no deferred rules.
    add-cross-product(fare-component-set1-without-rules, fare-component-set2-without-rules)
    // For the remainder of fare-components, though, explicit enumeration is necessary.
    enumerate-priceable-units(fare-component-set1-with-rules, fare-component-set2-without-rules)
    enumerate-priceable-units(fare-component-set1, fare-component-set2-with-rules)
    return(OR-AND-OR)
```

FACTORING PRICEABLE-UNITS OF SIZE THREE OR GREATER

Properly enumerating all possible combinations of fare-components for priceable-units of size three or greater is computationally burdensome, though possible.

Referring now to FIG. **15**, a preferred procedure **260** finds a subset of possible priceable-units. In particular, it extracts **262** those fare-components that have no deferred record-2s, and build priceable-units from them. Since there are no deferred record-2s, there are no intra-priceable-unit constraints and it is possible to construct a factored representation.

The procedure **268** that factors priceable-units of size three or greater, get-OR-AND-OR-3+, (TABLE 29) applies reevaluate-deferred-record-2s to each set of fare-components, to filter them. It then partitions the resulting sets based on surcharges, and takes the cross-product of these sets to construct the proper OR-AND-OR representation. The procedure for the last case does not return all possible valid priceable-units.

TABLE 29

```
get-OR-AND-OR-3+(faring-markets, fare-component-sets, environmental-information)
    Let-OR-AND-OR={ {} }
    For fare-component-set in fare-component-sets
        Let valid-fare-components = reevaluate-deferred-record-2s(faring-markets, fare-component-set,
                                                                  environmental-information)
        Let new-OR-AND-OR = {}
        For OR in partition-fare-components-by-surcharges(valid-fare-components)
            For previous-AND-OR in OR-AND-OR
                new-OR-AND-OR += postpend(previous-AND-OR, OR)
        OR-AND-OR = new-OR-AND-OR
    return(OR-AND-OR)
```

Priceable-units of size three or greater tend to have more deferred record-2s. This may somewhat reduce the effectiveness of the extracting procedure **262**. The prevalence of deferred record-2s rules occurs because in complicated journeys, time-bounds used in an initial rule application tend to be broadly specified.

At this stage of processing time bound ranges can be tightened, because the faring-markets that comprise the priceable-unit are known. Therefore, deferred record-2s can be reapplied **264** to faring-atoms in the same manner that they are applied in the initial rule application. If all deferred record-2s pass, then the faring-atom is retained **266**. If a record-2 fails or is deferred, the faring-atom is discarded. The function reevaluate-deferred-record-2s (TABLE 28) performs this filtering. It takes a set of faring-markets and a fare-component set, and sets time-bounds based on the faring-markets. It reevaluates deferred record-2s for each fare-component in the set, and returns the set of fare-components that have all their record-2s pass.

LINKING ITINERARIES

Priceable-units-labels associate faring-atoms from one or more slices with fares. In the pricing-graph representation **38'** set of pricing solutions, sets of priceable-unit-labels are used to link itineraries from different slices.

The pricing graph representation **38'** of pricing-solutions **38** is constructed by selecting a set of priceable-unit-labels. Each of these PU-labels may or may not project onto a given slice of a journey. For example, in a round-trip journey a round-trip priceable-unit-label will project onto both slices, while a one-way priceable-unit-label will project onto only one slice of the journey. Once a set of PU-labels has been chosen, in any slice any itinerary may be chosen so long as it has some division that has faring-atoms containing exactly the PU-labels that project onto that slice. The choice of itinerary is otherwise independent of choices made in other slices.

A set of PU-labels encodes all constraints that exist between itineraries in different slices. This leads to a rela-

TABLE 28

```
reevaluate-deferred-record-2s(faring-markets, fare-component-set, environmental-information)
    set-time-bounds(faring-markets)
    Let fare-components = {}
    For fare-component in fare-component-set
        Let result = pass
        Let deferred-record-2s = {}
        For record-2 in fare-component.deferred-record-2s
            Let record-2-result, record-2-surcharges =
                apply-record-2-FC(record-2, fare-component.faring-atom, environmental-information)
            If (record-2-result = pass)
                fare-component.surcharges += record-2-surcharges
            Else If (record-2-result = defer)
                deferred-record-2s += record-2
            Else
                result = fail
        If (result = pass)
            fare-component.deferred-record-2s = deferred-record-2s
            fare-components += fare-component
    return(fare-components)
```

tively simple procedure for constructing the pricing-graph. Itineraries within each slice are indexed by the sets of multi-slice PU-labels they can be associated with. These indices are called slice-label-sets, and act as a logical OR over itineraries. The slice-label-sets from different slices are linked by matching PU-labels.

Single-slice (one-way) priceable-unit-labels are treated somewhat differently than multi-slice priceable-unit-labels to enhance efficiency. In particular, there is no need to include single-slice PU-labels in slice-label-sets, because they do not need to be matched across slices. Rather, single-slice PU-labels are attached closely to itineraries in the pricing-graph. That is, within a slice-label-set, each itinerary is associated with a compact representation of the set of single-slice PU-labels that can be used with the itinerary, given that the multi-slice PU-labels found within the slice-label-set are also used.

The linking process constructs slice-label-sets with each slice-label-set being a set of multi-slice PU-labels and associated itinerary divisions. Slice-label-sets group itineraries by multi-slice PU-labels. Each division has associated with it a set of single-slice PU-labels.

In the pricing-graph, slice-label-sets act as ORs over itineraries. Multi-slice PU-labels encapsulate information concerning the itinerary to permit the linking process to operate over slice-label-sets rather than individual itineraries. This approval is computationally efficient and results in small pricing-graphs. In each slice-label-set, each itinerary (division) is paired with a set of single-slice PU-labels. During construction of the pricing graph, each slice-label-set is transformed into an OR over ANDs of itineraries and sets of PU-labels.

The linking process **282** also connects or links slice-label-sets from different slices, as shown in FIG. **16**. Connecting is accomplished by starting from the first slice and working forward to the last slice. Intermediate results are summarized in open-label-sets. Each open-label-set is a set of (multi-slice) PU-labels that project onto slices that have not been processed, along with a set of backward-links that are each a pair of a slice-label-set and an open-label-set from the previous slice. Processing starts retrieving **283** slice-1 and matching with a single, empty slice-0 open-label-set **288**. Slice-label-sets from slice 1 are "added" **290** to this open-label-set, resulting in new slice-1 open-label-sets. Then slice-label-sets from slice-2 are added to these, resulting in slice-2 open-label-sets, and so on. As this process continues, PU-labels that are complete **293** (i.e., that do not project to subsequent slices) are removed **293** from open-label-sets. If PU labels are not complete, the index of the slice is incremented **294** and the next slice is retrieved **287**. If any pricing-solutions exist, the process will terminate **299** in a single, empty, last-slice open-label-set determined at **291**. At that point, the backward-links serve as the top-level representation of the pricing-graph.

Set forth below is an example of the linking process. This example assumes a three-slice circle-trip journey, from BOS to MSP to MIA. In the following discussion, PU-labels are identified by a unique letter followed by a sequence of digits that indicate which slices the PU-label projects onto. For example, A-23 is a two-component PU-label that projects onto the second and third slices. Each itinerary may have several divisions, and each division may have many possible collections of PU-labels (with each collection built by selecting one PU-label per faring-atom in the division).

At this stage of processing in the faring process **18** divisions have been produced for each itinerary, each division comprising a set of faring-atoms. PU-labels have been constructed, and stored in each faring-atom. From this information it is possible to enumerate for every division, possible collections of PU-labels, by selecting one for each faring-atom. TABLE 30 below summarizes the result of this procedure, for the example journey. Each possible collection of PU-labels is partitioned into those that project onto only one slice (one-way priceable-units) and those that project onto more than one-slice. In this table, divisions are encoded using the endpoints of faring-atoms, to save space, and each itinerary and division are given numeric labels so that they can be referenced in the rest of the discussion.

TABLE 30

Slice	Itinerary	Division	Multi-Slice PU-Labels	Single-Slice PU-Labels
1	1.1 BOS→MSP UA123	1.1.1 BOS→MSP	{A-123}	{ }
			{F-13}	{ }
			{ }	{I-1}
1	1.2 BOS→CHI NW315 CHI MSP UA739	1.2.1 BOS→MSP; CHI→MSP	{B-13 C-12}	{ }
			{B-13}	{H-1}
			{C-12}	{G-1}
			{ }	{G-1 H-1}
1	1.3 BOS→MSP CO450	1.3.1 BOS→MSP	{ }	{J-1}
2	2.1 MSP→MIA UA901	2.1.1 MSP→MIA	{A-123}	{ }
			{ }	{K-2}
2	2.2 MSP→CBI UA623 CHI MIA UA841	2.2.1 MSP→MIA	{A-123}	{ }
			{ }	{K-2}
		2.2.2 MSP→CHI; CHI→MIA	{C-12 D-23}	{ }
			{C-12}	{M-2}
			{D-23}	{L-2}
			{ }	{L-2 M-2}
2	2.3 MSP→MIA FL207	2.3.1 MSP→MIA	{E-23}	{ }
3	3.1 MIA→BOS UA112	3.1.1 MIA→BOS	{A-123}	{ }
3	3.2 MIA→CHI UA487 CHI→BOS NW316	3.2.1 MIA→CHI; CHI→BOS	{D-23 B-13}	{ }
			{D-23}	{O-3}
			{B-13}	{N-3}
			{ }	{N-3 O-3}
3	3.3 MIA→MSP FL208 MSP→BOS UA558	3.3.1 MIA→MSP; MSP→BOS	{E23 F13}	{ }
			{E23}	{P-3}

As TABLE 30 above shows, there are three different itineraries for each slice. Each itinerary is split into one or two divisions of faring-atoms. Each division has one or several possible PU-label combinations. For example, for the second division of the second slice-2 itinerary (2.2.2) there are four different PU-label sets. This is because the division has two faring-atoms, and each faring-atom has two possible PU-labels. For reference, the table below lists each hypothetical PU-label along with its faring-markets.

TABLE 31

Name	Faring-Markets			Comment
A-123	1: UA BOS→MSP	2: UA MSP→MIA	3: UA MIA→BOS	3-Component Circle Trip
B-13	1: NW BOS→CHI		3: NW CHI→BOS	Round Trip
C-12	1: UA CHI→MSP	2: UA CHI→MSP		Round Trip
D-23		2: UA CHI→MIA	3: UA MIA→CHI	Round Trip
E-23		2: FL MSP→MIA	3: FL MIA→MSP	Round Trip
F-13	1: UA BOS→MSP		3: UA MSP→BOS	Round Trip
G-1	1: NW BOS CHI			One Way
H-1	1: UA CHI→MSP			One Way
I-1	1: UA BOS→MSP			One Way
J-1	1: CO BOS→MSP			One Way
K-2		2: UA MSP→MIA		One Way
L-2		2: UA MSP→CHI		One Way
M-2		2: UA CHI→MIA		One Way
N-3			3: UA MIA→CHI	One Way
O-3			3: NW CHI→BOS	One Way
P-3			3: UA MSP→BOS	One Way

TABLE 32 below lists slice-label-sets that are produced in this example, and as with itineraries and itinerary divisions, the faring process assigns each a numerical label. Many itineraries may be grouped into the same slice-label-set. For example, there are three different itinerary divisions that are grouped into slice-label-set 1.3. TABLE 30 lists, for each slice-label-set, its backward-projection. This is the set of multi-slice PU-labels that project onto previous slices.

TABLE 32

Slice	Multi-Slice PU-Labels	Itinerary	Division	Single-Slice PU-Labels	Backward Projection
1	1.1 {A-123}	1.1	1.1.1	{ }	{ }
1	1.2 {F-13}	1.1	1.1.1	{ }	{ }
1	1.3 { }	1.1	1.1.1	{I-1}	{ }
		1.2	1.2.1	{G-1 H-1}	
		1.3	1.3.1	{J-1}	
1	1.4 {B-13 C-12}	1.2	1.2.1	{ }	{ }
1	1.5 {B-13}	1.2	1.2.1	{H-1}	{ }
1	1.6 {C-12}	1.2	1.2.1	{G-1}	{ }
2	2.1 {A-123}	2.1	2.1.1	{ }	{A-123}
		2.2	2.2.1	{ }	
2	2.2 { }	2.1	2.1.1	{K-2}	{ }
		2.2	2.2.2	{L-2 M-2}	
2	2.3 {C-12 D-23}	2.2	2.2.2	{ }	{C-12}
2	2.4 {C-12}	2.2	2.2.2	{M-2}	{C-12}
2	2.5 {D-23}	2.2	2.2.2	{L-2}	{ }
2	2.6 {E-23}	2.3	2.3.1	{ }	{ }
3	3.1 {A-123}	3.1	3.1.1	{ }	{A-123}
3	3.2 {D-23 B-13}	3.2	3.2.1	{ }	{D-23 B-13}
3	3.3 {D-23}	3.2	3.2.1	{O-3}	{D-23}
3	3.4 {B-13}	3.2	3.2.1	{N-3}	{B-13}
3	3.5 { }	3.2	3.2.1	{N-3 C-3}	{ }
3	3.6 {E23 F13}	3.3	3.3.1	{ }	{E-23 F-13}
3	3.7 {E23}	3.3	3.3.1	{P-3}	{E-23}

Shown in the TABLE 33 below are lists for each slice of the open-label-sets, as well as their backward-links and their next-slice-projection. This last field is the subset of open PU-labels that project onto the subsequent slice. It is equal to the backward-projection of any slice-label-set that is part of a backward-link to the open-label-set.

TABLE 33

Slice	Open-Label-Set	Next-Slice Projection	Backward-Link Slice-Label-Set	Backward-Link Open-Label-Set
0	0.1 { }	{ }		
1	1.1 {A-123}	{A-123}	1.1 {A-123}	0.1 { }
1	1.2 {F-13}	{ }	1.2 {F-13}	0.1 { }
1	1.3 { }	{ }	1.3 { }	0.1 { }
1	1.4 {B-13 C-12}	{C-12}	1.4 {B-13 C-12}	0.1 { }
1	1.5 {B-13}	{ }	1.5 {B-13}	0.1 { }
1	1.6 {C-12}	{C-12}	1.6 {C-12}	0.1 { }
2	2.1 {A-123}	{A-123}	2.1 {A-123}	1.1 {A-123}
2	2.2 { }	{ }	2.2 { }	1.1 { }
2	2.3 {D-23}	{D-23}	2.3 {C-12 D-23}	1.6 {C-12}
			2.5 {D-23}	1.3 { }
2	2.4 {E-23}	{E-23}	2.6 {E-23}	1.3 { }
2	2.5 {D-23 F-13}	{D-23 F-13}	2.5 {D-23}	1.2 {F-13}
2	2.6 {E-23 F-13}	{E-23 F-13}	2.6 {E-23}	1.2 {F-13}
2	2.7 {F-13}	{F-13}	2.2 { }	1.2 {F-13}
2	2.8 {D-23 B-13}	{D-23 B-13}	2.3 {C-12 D-23}	1.4 {B-13 C-12}
			2.5 {D-23}	1.5 {B-13}
2	2.9 {E-23 B-13}	{E-23 B-13}	2.6 {E-23}	1.5 {B-13}
2	2.10 {B-13}	{B-13}	2.4 {C-12}	1.4 {B-13 C-12}
			2.2 { }	1.5 {B-13}
3	3.1 { }	{ }	3.1 {A-123}	2.1 {A-123}
			3.5 { }	2.2 { }
			3.3 {D-23}	2.3 {D-23}
			3.7 {E-23}	2.4 {E-23}
			3.6 {E-23 F-13}	2.6 {E-23 F-13}
			3.2 {D-23 B-13}	2.8 {D-23 B-13}
			3.4 {B-13}	2.10 {B-13}

Each open-label-set contains a set of PU-labels that are still "open", i.e., project onto a subsequent slice. For example, the PU-label C-12 does not appear in open-label-sets from slice 2 or slice 3. In the pricing-graph, each open-label-set will be translated into an OR over the

backward-links. The backward-links represent paths that lead to the open-label-set. Each is a pair (an AND) of a slice-label-set with an open-label-set from the previous slice. Because TABLE 33 is consistent, the backward-projection of any slice-label-set in a link is equal to the next-slice-projection of the open-label-set in the link. Furthermore, the PU-labels in each open-label-set can be constructed by selecting any backward-link, taking the union of the PU-labels in the link's open-label-set and slice-label-set, and removing any PU-labels that do not project forward.

If there is an empty open-label-set for the last slice, then pricing-solutions exist. This open-label-set provides the "root" of the pricing-graph, a node that has a child for every link. Each of these links will become an AND of the slice-label-set and the previous open-label-set. In this way open-label-sets and slice-label-sets are used to produce the pricing-graph.

FARE-COMBINABILITY RESTRICTIONS

The linking procedure described above assumes that there are no restrictions on the mixing of priceable-unit-labels other than those imposed by itineraries. This is not always the case. For example, although the various create-PUs-in-markets procedures described above apply fare-combinability checks, those checks include only restrictions on the fares that may coexist within a priceable-unit. These checks include ATPCO categories 101, 102 and 103, but not category 104. The category-10 record-2s that are stored on fare-components may also include so called "end-on-end" fare-combinability restrictions. These restrictions constrain the fares within other priceable-units. One example of such an end-on-end fare-combinability constraint is that all fares within an entire journey must be published by the same airline as the fare with the constraint.

Cross-priceable-unit constraints such as end-on-end fare-combinability restrictions complicate the process of finding valid fares for itineraries. In general cross-priceable unit constrains can be very expensive to evaluate. These constraints can often be efficiently evaluated during the process that links the set of valid fares to the sets of flights to form a pricing solution. Below, an efficient approach for evaluating many common end-on-end fare-combinability restrictions is described.

First, priceable-unit-labels are constructed in such a manner that all priceable-unit-cores within them share the same end-on-end combinability restrictions. This is reflected in the procedure partition-fare-components-into-sets, described previously. During the linking process each priceable-unit-label end-on-end fare-combinability restriction is applied to the fares in other priceable-unit-labels. This happens during the initial stage of the linking process, in the execution of create-slice-label-sets. Create-slice-label-sets iterates over itinerary divisions, and for each division, iterates in an inner loop over covering sets of priceable-unit-labels. In this inner loop an end-to-end fare-combinability restriction attached to one priceable-unit-label in the set can be applied to fares in other priceable-unit-labels within the set. If the restriction fails, the set of priceable-unit-labels is rejected.

For this procedure it is desirable that every priceable-unit-label containing an end-on-end restriction projects onto all the slices that the restriction needs to be checked against. Some restrictions may only need to be checked against fares adjacent to the fare-component or priceable-unit containing the restriction, while others may need to be applied to every other priceable-unit in the entire journey. Hence, if a priceable-unit-label projects onto every slice, then its end-on-end restrictions can be applied to every other priceable-unit-label in a potential pricing-solution.

But, if a priceable-unit-label projects onto only one slice (as a one-way priceable-unit-label would) while its restriction must be applied to priceable-units from several slices, then the restriction cannot be applied using the method described here. In such a case there are several options available. One is to reject the set of priceable-unit-labels currently under consideration. Another is to accept it, but mark it as potentially requiring that any solution containing it must be split into several journeys (a "split ticket").

When a combinability record-2 is applied in this manner, the information available to it at any one time is a set of priceable-unit-labels that collectively cover a division of an itinerary (one of these priceable-unit-labels contains the source record-2). Each of these priceable-unit-labels reference one or more priceable-unit-cores, each of which in turn references one or more fares. It is these fares that are validated by the combinability record-2. It may be that some priceable-unit-cores from within a priceable-unit-label pass, while others fail. Several options are available in this ambiguous case: a new priceable-unit-label can be generated, referencing only those priceable-unit-cores that pass, or the entire priceable-unit-label can be rejected. The second is the more efficient option, though it may cause some valid pricing-solutions to be unnecessarily rejected.

CONSTRUCTING SLICE-LABEL-SETS

Slice-label-sets are constructed during the process of producing open-label-sets **282**, by the pseudo-code given below (TABLE 34). This code is passed the set of itineraries for a slice.

For each itinerary and each division within that itinerary, all possible sets of PU-labels are enumerated. Each set is partitioned into a set of multi-slice PU-labels and a set of single-slice PU-labels. A unique slice-label-set is produced for each collection of multi-slice PU-labels. Within the slice-label-set are stored itinerary-label-holders. Each of these pairs an itinerary with a set of division-label-holders. Each division-label-holder pairs an itinerary division with a set of sets of single-slice PU-labels.

TABLE 34

```
create-slice-label-sets(itineraries)
    Subroutine division-PU-label-sets(division)
        //
        // Return a list of all the possible sets of PU-labels for this division.
        //
        Let PU-label-sets = { {} }
        For faring-atom in division
            Let new-PU-label-sets = { }
            For PU-label in faring-atom.priceable-unit-labels
                For previous-PU-label-set in previous-PU-label-sets
```

TABLE 34-continued

```
                new-PU-label-sets += postpend(previous-PU-label-set, PU-label)
            PU-label-sets = new-PU-label-sets
        return(PU-label-sets)
    Let slice-label-sets = { }
    For itinerary in itineraries
        For division in itinerary.divisions
            For PU-labels in division-PU-label-sets(division)
                Let multi-slice-PU-labels = multi-slice-PU-labels(PU-labels)
                Let single-slice-PU-labels = single-slice-PU-labels(PU-labels)
                Let slice-label-set = find(multi-slice-PU-labels, slice-label-sets)
                If (slice-label-set = nil)
                    slice-label-set = new-slice-label-set( )
                    slice-label-set.multi-slice-PU-labels = multi-slice-PU-labels
                    slice-label-set.division-single-slice-labels = { }
                Let itinerary-label-holder = find(itinerary, slice-label-set.itinerary-label-holders)
                If (itinerary-label-holder = nil)
                    itinerary-label-holder = new-itinerary-label-holder( )
                    itinerary-label-holder.itinerary = itinerary
                    itinerary-label-holder.division-label-holders = { }
                    slice-label-set.itinerary-label-holders += itinerary-label-holder
                Let division-label-holder = find(division, itinerary-label-holder.division-label-holders)
                If (division-label-holder = nil)
                    division-label-holder = new-division-label-holder( )
                    division-label-holder.division = division
                    division-label-holder.single-slice-label-sets = {}
                    itinerary-label-holder.division-label-holders += division-label-holder
                division-label-holder.single-slice-label-sets += single-slice-PU-labels
    return(slice-label-sets)
```

CONSTRUCTING OPEN-LABEL-SETS

Open-label-sets are constructed slice-by-slice, starting from a single, empty open-label-set. For each slice, the first step is to produce slice-label-sets using the procedure described above and enumerate the current set of open-label-sets. For each open-label-set, the projection into the next slice is computed. All slice-label-sets with backward-projections equal to that next-slice-projection are used to create a new set of open multi-slice priceable-unit-labels, and from this a new open-label-set is produced. The pseudo code below in TABLE 35 describes this.

TABLE 35

```
link-itineraries(itinerary-sets)
    Subroutine projection(PU-labels, slice-number)
        //
        // Return those PU-labels that project onto this slice.
        //
        Let result = { }
        For PU-label in PU-labels
            If (find(slice-number, PU-label.slice-numbers))
                result += PU-label
        return(result)
    Subroutine backward-projection(PU-labels, slice-number)
        //
        // Return those PU-labels that project onto a previous slice.
        //
        Let result = {}
        For PU-label in PU-labels
            If (find-if-less-than(slice-number, PU-label.slice-numbers))
                result += PU-label
        return(result)
    Subroutine forward-projection(PU-labels, slice-number)
        //
        // Return those PU-labels that project onto a subsequent slice.
        //
        Let result = {}
        For PU-label in PU-labels
            If (find-if-greater-than(slice-number, PU-label.slice-numbers))
                result += PU-label
        return(result)
    Let initial-open-label-set = new-open-label-set( )
    initial-open-label-set.open-PU-labels = {}
    initial-open-label-set.backward-links = {}
    Let open-label-sets = list(initial-open-label-set)
            Let open-PU-labels open-label-set.open-PU-labels
            Let next-slice-projection = projection(open-PU-labels, slice-number)
            For slice-label-set in slice-label-sets
            Let slice-PU-labels = slice-label-set.multi-slice-PU-labels
```

TABLE 35-continued

```
            If (next-slice-projection = backward-projection(slice-PU-labels, slice-number))
            Let new-open-PU-labels = forward-projection(union(slice-PU-labels, open-PU-labels),
                                                        slice-number)
                Let new-open-label-set = find(new-open-PU-labels, new-open-label-sets)
                If (new-open-label-set = nil)
                    new-open-label-set = new-open-label-set( )
                    new-open-label-set.open-PU-labels = new-open-PU-labels
                    new-open-label-set.backward-links = {}
                Let backward-link = new-backward-link( )
                backward-link.slice-label-set = slice-label-set
                backward-link.open-label-set = open-label-set
                new-open-label-set.backward-links += backward-link
        open-label-sets = new-open-label-sets
        slice-number += 1
If (open-label-sets = {})
        return(nil)
    Else
        // Return the root open-label-set.
        return(first(open-label-sets))
```

PRICING GRAPH

Referring now to FIG. **17**, a pricing graph **38'** (FIG. **3**) is produced containing logically combinable nodes that can be used to efficiently and compactly represent a set of pricing solutions **38** (FIG. **1**). The pricing graph **38'** as used herein is a so-called directed acyclic graph although other types of representations could be used. For example, a grammar could be used.

The pricing graph **38'** is constructed **300** from data structures **300***a* (summarized below in TABLE 36 and mentioned in conjunction with FIGS. **4A–4B**) provided during the preceding processes. The data structures convert **300***b* to one or more nodes in the pricing graph. The open-label-set data structure becomes an OR node and its children are the converted versions of its backward links. Each backward link in the open label set is converted to an AND node. If a pricing object is shared, for example, a priceable unit core is shared between several priceable unit labels, then its counterpart nodes in the pricing graph will be shared so that the size of the pricing graph is not unnecessarily enlarged. The converted nodes are placed **300***c* in the pricing graph nodes. Terminal objects such as fares and itineraries undergo essentially no conversion. They are placed **300***d* into special terminal nodes with no children and are obtained from the various data structures that have the pricing objects.

TABLE 36

Data-Structure Type	Type	Child Fields
open-label-set	OR	backward-links (each a backward-link).
backward-link	AND	open-label-set (an open-label-set). slice-label-set (a slice-label-set).
slice-label-set	AND (OR)	multi-slice-PU-labels (each a priceable-unit-label). itinerary-label-holders (each an itinerary-label-holder). The itinerary-label-holders are put into an OR and the OR is placed under the AND, which also includes all multi-slice-PU-labels.
itinerary-label-holder	AND (OR)	itinerary (an itinerary). division-label-holders (each a division-label-holder). The division-label-holders are put into an OR and the OR is placed in an AND with the itinerary.
division-label-holder	OR (AND)	single-slice-PU-label-sets (each a set of sets of PU-labels). The inner sets are put into ANDs, and these are all embedded under an OR.
priceable-unit-label	OR	priceable-unit-cores (each a priceable-unit-core).
priceable-unit-core	AND	fares (each a fare). surcharges (each a surcharge, penalty, discount, etc.)
fares	Term	
itinerary	Term	
surcharge	Term	

In cases where a node has only one child, there is no need to produce the node. Rather, a direct link can be passed to its child. This does not alter the interpretation of the pricing-graph, but can result in a smaller graph.

The pseudo-code below TABLE 37 summarizes construction of the pricing graph, given the "root" open-label-set that is the output of the linking process.

TABLE 37

```
create-pricing-graph(root-object)
    Let nodes = {}
    Subroutine convert-object(object)
        // Convert the object and cache the result, so that nodes with multiple parents are shared
        Let node = find(object, nodes)
        If (node = nil)
            node = convert(object)
            nodes += node
        return(node)
    Subroutine convert-objects (objects)
        // Convert the set of objects and return a set of nodes.
        Let result = {}
        For object in objects
            result += convert-object(object)
        return(result)
    Subroutine create-node(children, type)
        // Create a node of type type with children children, if there is more than one child.
        // Otherwise the node is unnecessary, and the only child is returned.
        If (length(children) = 1)
            return(first(child)
        Else
            Let node = new-node( )
            node.type = type
            node.children = children
            node.terminal-object = nil
            return(node)
                        The pricing graph 38' resulting from the search procedure 54
Subroutine convert(object)
    Let object-type = type(object)
    If (type = open-label-set)
        return(create-node(convert-objects(object.backward-links), OR))
    Else If (type = backward-link)
        return(create-node(list(convert-object(object.slice-label-set), convert-object(object.open-label-set)), AND)
    Else If (type = slice-label-set)
        Let children = convert-objects(object.multi-slice-PU-labels)
        children += create-node(convert-objects(object.itinerary-label-holders), OR)
        return(create-node(children, AND))
    Else If (type = itinerary-label-holder)
        Let children = convert-objects(object.division-label-holders)
        children += create-node(convert-objects(object.itinerary), OR)
        return(create-node(children, AND))
    Else If (type = division-label-holder)
        Let children = {}
        For single-slice-PU-label-set in object.single-slice-PU-label-sets
            children += create-node(convert-objects(single-slice-PU-label-set), AND)
        return(create-node(children, OR))
    Else If (type = priceable-unit-label)
        return(create-node(convert-objects(object.priceable-unit-cores), OR)
    Else If (type = priceable-unit-core)
        return(create-node(append(convert-objects(object.fares), convert-objects(object.surcharges)), AND)
    Else // object is a terminal.
        Let node = new-node( )
        node.type = terminal
        node.children = {}
        node.terminal-object = object
        return(node)
return(convert-object(root-object))
```

provides a compact way for representing a very large number of set of pricing solutions. By the above process, it is often possible to obtain a very large number of pricing solution components. Although the number of pricing solutions can be returned in the form of a simple list, this is not desirable. A very large number of pricing solutions can be difficult to manipulate, enumerate and interrogate and to transfer/transmit across a network since the amount of data involved is very large. The pricing graph **38'** provides a more compact way of representing these pricing solutions. The compact representation of the range of set of pricing solutions is generated where choices are represented explicitly and redundancy is removed wherever possible.

As mentioned above, the pricing graph **38'** produced by the search procedure **54** includes three types of nodes. The first type of node is a node that represents choices called "LOGICAL OR" nodes. The second type of node is a node that represents collections referred to as "LOGICAL AND" nodes. A third type of node represented in the pricing graph is a terminal node that represents pricing objects.

A data structure representation (TABLE 38) of the nodes is set out below. Each node contains a "type", which specifies whether the node is an AND node, or OR node or a terminal node. The data structure also contains either list of children (if the node is an AND node or an OR node) or

a terminal object (if the node is a terminal). The node contains fields that store values used by algorithms that manipulate the pricing graph **38'**.

TABLE 38

Node fields	Use
type	Either AND, OR or TERMINAL
children	A list of child-nodes, if node is AND or OR.
terminal-object	A terminal-object such as a fare or itinerary, if the node is TERMINAL.
active?	True or false, depending on whether the node is active.
inner-value	The node's minimum possible inner-value.
outer-value	The node's minimum possible outer-value.
total-value	The node's minimum possible total-value.
best-child	For OR-nodes, the child with the least-positive inner-value.

As mentioned above, the pricing graph **38'** is a compact representation of a set of set of pricing solutions. The typical number of set of pricing solutions represented by pricing graph ranges from tens of millions into hundreds of billions with the number of nodes in the graph ranging from thousands to tens of thousands. The pricing graph can be easily stored and/or transmitted over a network or other connection to a client and represents complete representation of all or substantially all of possible pricing solutions. Therefore, the pricing graph **38'** can be used by a smart client without further intervention from the server **12**.

MANIPULATING THE PRICING-GRAPH

Referring now to FIG. **18**, a high level illustration of a process **301** that operates on the pricing graph **38'** typically as a client process **36** on the client computer system is shown. The process **301** includes a user query **302** that passes parameters into a process **304** and a value function **306** to extract from the pricing graph **38'** certain pricing solutions **308** that satisfy parameters specified by the user query **302**. The extracted pricing solutions are returned as a list **308** or other representation. Generally the pricing solutions are displayed on the display **40**. Display software **309** handles production of GUI's **41** to present the information.

The pricing solution list **308** will contain pricing solutions extracted from the pricing graph **38'** in accordance with user specified parameters from the user query **302** using one of the processes **304** and one of the value functions **306**.

Referring now to FIG. **19**, illustrative processes are shown. In particular, in response to the user query **310**, one of the processes is executed. The processes **312** can comprise a find best "value" and pricing solutions associated with the value (e.g., price) process **312***a*; find best "value" and pricing solutions associated with the value for "node" (e.g., find best price for a particular itinerary) process **312***b*; an enumeration for "N" pricing solutions **312***c*; or an enumeration process that lists the pricing solutions according to some "value."Additional enumeration processes can be provided to produce query results corresponding to different ways of looking at pricing solutions extracted from the pricing graph **38'**. A node invalidating process **314** that invalidates selected nodes from contributing to a pricing solution is also included.

Examples of each of these processes are set forth below.

Efficient algorithms **312** are used for manipulating this representation to extract information of interest and to enumerate set of pricing solutions from the structure. For example, it is possible to quickly extract the cheapest solution; to find the cheapest solution involving selected fares and itineraries; to verify whether any pricing solution remains if specific fares or itineraries are excluded; to enumerate solutions under various orderings and so forth. Furthermore, the representation is compact enough so that it can be efficiently stored and transmitted such as from the server to the client. One benefit, therefore, is that after a single fare search **54** in the server process **16**, the server process **16** transfers the pricing graph **38** to the client process **36**. The client process **36** can examine and manipulate the large space of pricing solutions represented in the pricing graph **38'** without further interaction with the server process **18**.

For the set of pricing solutions represented by the pricing graph **38'** to be useful, processes are provided to extract pricing solutions from the graph and manipulate the set of pricing solutions, as depicted in FIG. **19**. In general, each of the enumeration processes to be described operate on the pricing graph to extract pricing solutions from the pricing graph according to particular criteria that can be set, for example, by a client system **30***c* (FIG. **2**) in response to a user query **48** (FIG. **4**). Examples of user queries as well as a display representation for information extracted from the pricing graph will be described below.

An example of an enumeration function enumerates pricing solutions in a specific order. For example, an enumeration function can enumerate the 100 cheapest pricing solutions represented by the pricing graph **38'**. A second enumeration function can find extreme points of the set of pricing solutions. This can be used, for example, to find the most convenient pricing solution. In addition, a value function can specify a minimum value of some value over the set of pricing solutions that involve a particular node. One value function finds for each itinerary the cheapest pricing solution that involves that itinerary or the shortest total duration of any pricing solution that involves that itinerary.

In addition, each of the above operations can be performed on a subset of the graph. For example, it may be desirable to enumerate the 100 cheapest solutions that involve a given itinerary or finding the most convenient solution that involves only refundable fares or includes only certain airlines or excludes certain airlines.

VALUE FUNCTIONS

Referring now to FIG. **19**, many processes or operations on the pricing graph **38'** use a value-function, a function that operates on the terminal nodes of the pricing graph **38'** and returns a numerical value that can be used to rank pricing-solutions. Examples of value-functions include price computed by summing the prices of fares (and penalties and surcharges) in a pricing-solution, duration, or convenience (that might be a mix of total travel-time with penalties for stops and airline-changes, for example), or mixes of each.

Many of the processes used to manipulate the pricing graph **38'** depend on a value-function being decomposable into the sum of a second function that is applied to individual terminal nodes in the pricing-graph. The "price value function" meets this requirement, because the total price of a pricing-solution is equal to the sum of the prices of fares. Many expressions of convenience also meet this requirement, including those that can be computed as the sum of a function applied to individual itineraries. However, there are some value-functions that cannot be placed into this form. An example is a "convenience" function that checks whether travel in different slices is on the same airline. Such a function depends on all itineraries at once.

In general, in the discussion below, the term node-value-function is used to refer to a function that is applied to individual nodes in the pricing-graph, and summed to produce the value of an entire itinerary. The term value-function is used for the more general case of a function that may or

may not be decomposable into the sum of a node-value-function applied to each terminal in the pricing-graph.

FINDING THE BEST PRICING SOLUTION

The first process **312***a* is an example of one that finds extreme points of the set of pricing-solutions, such as the cheapest pricing-solution.

Assuming that it is desired to find a pricing-solution that minimizes some value-function that can be decomposed into a node-value-function F, the best pricing solution could be found by enumerating all pricing-solutions and applying F to each of them. This is impractical because of the large number of set of pricing solutions.

The Best Price algorithm **312***a*' efficiently finds the cheapest (best) price by starting at the "bottom" of the pricing-graph **38**' and constructing the best solution for each node by looking at the best solution of its children. In this way it works in one pass from the bottom of the graph to the top. At the end of the process the root node contains the best pricing solution for the entire pricing graph **38**.

The algorithm proceeds as follows: first, the nodes in the graph are ordered by depth and placed in a list, so that iterating over the list ensures that a child node is always encountered before its parent(s). Then, iterating across the list, the best value of F is computed for each node, using the already-computed values of F for its children. At this point every node in the graph is marked with its inner-value. The inner-value of a node is the best possible value of the function F on the set of (partial) pricing-solutions represented by the node. As inner-values are computed, for every OR node the child with the lowest inner-value is computed and stored. Finally, the best pricing solution can be constructed by starting at the root of the graph and collecting children. Whenever an OR node is encountered, the best child is chosen (the child with the lowest inner-value).

If a node is a terminal fare or itinerary, then its inner-value is the value of F applied to the node. If the node is an AND, representing a combination, then the minimum value of F over the partial solutions it represents is the sum of the minimum values of F over the partial solutions represented by each of its children. If a node is an OR, representing a choice, then the minimum value of F over the partial solutions it represents is found by making the optimal choice of children, that is, the child with the minimum inner-value. So the inner-value of an OR is the minimum of the inner-values of its children.

The pseudo-code in TABLE 38 summarizes the computation of inner-values. The function sort-nodes takes a root node and returns a list of all nodes under it, sorted by depth with the root node at the end. The procedure compute-inner-values takes in a sorted list of nodes as would be produced by sort-nodes, and a node-value-function. The procedure find-optimal-solution takes in a root-node and a node-value-function, calls sort-nodes and compute-inner-values to calculate inner-values for all nodes in the pricing-graph, and constructs a pricing-solution.

TABLE 38

```
sort-nodes(node)
    Let nodes = {}
    For child in node.children
            nodes = append(nodes, son-nodes(child))
    nodes += node
    return(nodes)
            compute-inner-values(sorted-nodes,
            node-value-function)
For node in sorted-nodes
    If (node.type = terminal)
        node.inner-value = apply(node-value-function,
        node.terminal-object)
    Else If (node.type = OR)
        node.inner-value = infinity
        For child in node.children
            If (child.inner-value < node.inner-value)
                node.inner-value = child.inner-value
                node.best-child = child
    Else If (node.type = AND)
        node.inner-value = 0
        For child in node.children
            node.inner-value += child.inner-value
find-optimal-solution(root-node, node-value-function)
    Subroutine get-node-solution(node)
        If (node.type = terminal)
            return(list(node.terminal-object))
        Else If (node.type = OR)
            return(get-node-solution(node.best-child)
        Else If (node.type = AND)
            Let solution = {}
            For child in node.children
                solution = append(solution, get-node-solution(child))
            return(solution)
    compute-inner-values(sort-nodes(root-node), node-value-function)
    return(get-node-solution(root-node))
```

FINDING MINIMUM VALUE

Another procedure **312***b* finds, for each node, the best (i.e., minimum) value of some value-function over all the set of pricing solutions involving that node. Another procedure **312***b*' finds for the best value, e.g., price and/or solutions involving "none" of the nodes. Price function **312***b* finds for each itinerary, the cheapest price of any pricing solution that contains that itinerary. These values can be computed efficiently, if the value-function can be decomposed into a node-value-function.

The best price value function **306***a* computes inner-values, as above, and computes for every node, an outer-value, equal to the minimum value contributed by all parts of the graph except that represented by the node. For each node, the minimum value of the value-function over all solutions that involve the node, (i.e., the total-value) is computed as the sum of that node's inner-value and outer-value.

The outer-value and total-value of a node are computed in a manner very similar to the computation of the inner-value. In particular, the outer-value for each node is calculated starting from the root of the graph, that has an outer-value of 0. Each node propagates outer-values down to its children. An OR-node passes its outer-value unchanged. An AND-node adds to its outer-value the inner-values of all children except that being propagated to. At every node, after the outer-value has been computed, the total-value is computed as the sum of the inner-value and outer-value.

When outer-values are propagated from a node to its children, a minimum computation is performed. This is because each child may have more than one parent, and its outer-value must be the minimum outer-value contributed by any parent. See TABLE 39 below.

TABLE 39

```
compute-outer-and-total-values(root-node, node-value-function)
        // Sort nodes and computer inner-values.
        Let sorted-nodes = sort-nodes(root-node)
        compute-inner-values(sorted-nodes, node-value-function)
        Let reversed-nodes = reverse(sorted-nodes)
    // Initialize all nodes to have outer-values of infinity.
    For node in reversed-nodes
        node.outer-value = infinity
    // The root-node has an outer-value of 0.
    first(reversed-nodes).outer-value = 0
    For node in reversed-nodes
        // Compute the total-value for the node.
        node.total-value = node.inner-value + node.outer-value
        If (node.type = OR)
            // For OR nodes, the outer-value is propagated down to its children unchanged.
            For child in node.children
                child.outer-value = minimum(child.outer-value, node.outer-value)
        Else If (node.type = AND)
            // For AND nodes, the outer-value is propagated down by adding the inner-values of
            // all but the child being propagated to. This is equal to the node's inner-value minus
            // the child's inner-value, which is equal to the node's total-value minus the child's
            // inner-value.
            For child in node.children
                child.outer-value = minimum(child.outer-value, node.total-value - child.inner-value)
```

INVALIDATING NODES

It may be desirable to "invalidate" 314 certain nodes from the pricing-graph **38**'. For example, itineraries containing or not containing specified airlines could be marked as not participating in the above algorithms, enabling the algorithms to find the best solutions involving or not involving these itineraries. The above algorithms can be easily adapted to accommodate checking whether the node is valid. In particular, the computation of inner-values, the first step in all the above algorithms, is modified to mark for every node whether the node represents any valid partial pricing-solutions given a specific query parameter. This information can be used in the rest of the algorithms. Every terminal node contains a field "valid?" that is either true or false. The compute-inner-values procedure uses these values to set the "valid?" field for non-terminals. See TABLE 40 below:

TABLE 40

```
// Initialize all nodes to have outer-values of infinity.
For node in reversed-nodes
    node.outer-value = infinity
// The root-node has an outer-value of 0.
first(reversed-nodes).outer-value = 0
For node in reversed-nodes
    If (node.valid? = true)
        // Compute the total-value for the node.
        node.total-value = node.inner-value + node.outer-value
        If (node.type = OR)
            // For OR nodes, the outer-value is propagated down to its children unchanged.
            For child in node.children
                child.outer-value = minimum(child.outer-value, node.outer-value)
        Else If (node.type = AND)
            // For AND nodes, the outer-value is propagated down by adding the inner-values of
            // all but the child being propagated to. This is equal to the node's inner-value minus
            // the child's inner-value, which is equal to the node's total-value minus the child's
            // inner-value.
            For child in node.children
                child.outer-value = minimum(child.outer-value,
                                    node.total-value - child.inner-value)
sort-nodes (node)
If (not(find(node,nodes)))
For child in node.children
sort-nodes-inner(child)
nodes += node
sort-nodes-inner(node):
return (nodes)
compute-inner-values(sorted-nodes.node-value-function)
    For node in sorted-nodes
        If (node.type = terminal)
            node.inner-value = apply(node-value-function, node.terminal-object)
        Else If (node.type = OR)
            node.inner-value = infinity
            node.valid? = false
            For child in node.children
                If (child.valid? = true and child.inner-value < node.inner-value)
                    node.inner-value = child.inner-value
```

TABLE 40-continued

```
                    node.best-child = child
                    node.valid? = true
            Else If (node.type = AND)
                node.inner-value = 0
                node.valid? = true
                For child in node.children
                    If (child.valid? = true)
                        node.inner-value += child.inner-value
                    Else
                        node.valid? = false
find-optimal-solution(root-node, node-value-function)
    Subroutine get-node-solution(node)
        If (node.type = terminal)
            return(list(node.terminal-object))
        Else If (node.type = OR)
            return(get-node-solution(node.best-child)
        Else If (node.type = AND)
            Let solution = {}
            For child in node.children
                solution = append(solution, get-node-solution(child))
            return(solution)
    compute-inner-values(sort-nodes(root-node), node-value-function)
    If (root-node.valid? = true)
        return(get-node-solution(root-node))
    Else
        return(nil)
compute-outer-and-total-values(root-node, node-value-function)
    // Sort nodes and computer inner-values.
    Let sorted-nodes = sort-nodes(root-node)
    compute-inner-values(sorted-nodes, node-value-function)
    Let reversed-nodes = reverse(sorted-nodes)
        // Initialize all nodes to have outer-values of infinity.
        For node in reversed-nodes
            node.outer-value = infinity
        // The root-node has an outer-value of 0.
        first(reversed-nodes).outer-value = 0
        For node in reversed-nodes
            If (node.valid? = true)
                // Compute the total-value for the node.
                node.total-value = node.inner-value + node.outer-value
                If (node.type = OR)
                    // For OR nodes, the outer-value is propagated down to its children unchanged.
                    For child in node.children
                        child.outer-value = minimum(child.outer-value, node.outer-value)
                Else If (node.type = AND)
                    // For AND nodes, the outer-value is propagated down by adding the inner-values of
                    // all but the child being propagated to. This is equal to the node's inner-value minus
                    // the child's inner-value, which is equal to the node's total-value minus the child's
                    // inner-value.
                    For child in node.children
                        child.outer-value = minimum(child.outer-value,
                                                    node.total-value - child.inner-value)
```

ENUMERATING PRICING SOLUTIONS

It is often desirable to arbitrarily enumerate many pricing solutions: the best, the second-best, the third-best, etc.

The enumeration algorithm **312***c* maintains a queue of partial-solutions, ordered by the lowest possible total value (e.g., price **312***c*) of the value-function over all complete solutions that contain the partial-solution. At the start of the search, a single partial solution is constructed from the root node of the pricing-graph **38'**. At each step the best partial-solution is dequeued, and expanded. Each partial-solution has a set of non-terminal nodes and a set of terminal objects. A partial-solution is expanded by selecting a non-terminal node and substituting the node's children (all of its children in the case of an AND, one of its children in the case of an OR). If a dequeued partial-solution contains only terminal objects, it is complete, and is returned. This process continues until the desired number of pricing-solutions that can be specified by a user has been produced.

The algorithm can accommodate value-functions that cannot be decomposed into the sum of a node-value-function. It does this by applying a second penalty-value-function to partial pricing-solutions as it constructs them. This function returns a non-negative number when given a new terminal object and existing set of terminal objects. The number is added to the values produced by the normal node-value-function. If the number is positive, it acts as a penalty. An example of how this could be used is for the case where a penalty is applied if travel in two slices is on different airlines. The penalty-value-function would return a (positive) penalty if the terminal was an itinerary, and the set of existing terminals contained an itinerary with travel on different airlines. Otherwise it would return 0. See TABLE 41 below.

TABLE 41

```
compute-inner-values(sorted-nodes, node-value-function)
    For node in sorted-nodes
        If (node.type = terminal)
            node.inner-value = apply(node-value-function, node.terminal-object)
        Else If (node.type = OR)
            node.inner-value = infinity
            node.valid? = false
            For child in node.children
                If (child.valid? = true and child.inner-value < node.inner-value)
                    node.inner-value = child.inner-value
                    node.best-child = child
                    node.valid? = true
            Else If (node.type = AND)
            node.inner-value = 0
            node.valid? = true
            For child in node.children
                If (child.valid? = true)
                    node.inner-value += child.inner-value
                Else
                    node.valid? = false
find-optimal-solution(root-node, node-value-function)
    Subroutine get-node-solution(node)
        If (node.type = terminal)
            return(list(node.terminal-object))
        Else If (node.type = OR)
            return(get-node-solution(node.best-daughter)
        Else If (node.type = AND)
            Let solution = {}
            For child in node.children
                solution = append(solution, get-node-solution(child))
            return(solution)
    compute-inner-values(sort-nodes(root-node), node-value-function)
    If (root-node.valid? = true)
        return(get-node-solution(root-node))
    Else
        return(nil)
compute-outer-and-total-values(root-node, node-value-function)
    // Sort nodes and computer inner-values.
    Let sorted-nodes = sort-nodes(root-node)
    compute-inner-values(sorted-nodes, node-value-function)
    Let reversed-nodes = reverse(sorted-nodes)
    // Initialize all nodes to have outer-values of infinity.
    For node in reversed-nodes
        node.outer-value = infinity
    // The root-node has an outer-value of 0.
    first(reversed-nodes).outer-value = 0
    For node in reversed-nodes
        If (node.valid? = true)
            // Compute the total-value for the node.
            node.total-value = node.inner-value + node.outer-value
            If (node.type = OR)
                // For OR nodes, the outer-value is propagated down to its children unchanged.
                For child in node.children
                    child.outer-value = minimum(child.outer-value, node.outer-value)
            Else If (node.type = AND)
                // For AND nodes, the outer-value is propagated down by adding the inner-values of
                // all but the child being propagated to. This is equal to the node's inner-value minus
                // the child's inner-value, which is equal to the node's total-value minus the child's
                // inner-value.
                For child in node.children
                    child.outer-value = minimum(child.outer-value,
                                                node.total-value - child.inner-value)
```

Referring now to FIG. **20**, a window **350** that is part of a graphical user interface of the client process **36** (FIG. **3**) is shown. The graphical user interface permits the user to access inter alia the enumeration processes **304**, value functions **306** and invalidate routine **307**. The graphical user interface has user selectable controls **352** such as "origin" and "destination". There are also controls for selecting time and date. In addition, there are controls **352** that specify "slices" of a journey. The user enters values corresponding to origin, destination, dates and time by selecting an appropriate one of the controls. The origin and destination controls **352** invoke a query window (FIG. **21**) where the user can enter a value. After the origin destination and preferably date and time information are entered, a user control "GO" becomes activated. Pressing the "GO" button will, for an initial query, send the query to the server process **18**. The server process handles the query and initiates the server process **18**. The server process **18** returns to the client process **36** a set of pricing solutions in a compact representation such as the pricing graph **38'**.

Referring now to FIG. **21**, a query entry window **360** is shown. The query entry window **360** is produced by activating either the ORIGIN or DESTINATION controls **352** in the window **350**. The query window **360** includes a user entry area **361** for entering a destination code or origin code (as appropriate) such as a three letter airport code or a

location such as a city, region or country (e.g., Turkey). Region **364** depicts a listing of airports in a region about the location entered in area **361**, whereas area **364** lists origins and destinations of a flight slice (none shown) that has been selected for the query. In addition, in the entry area **361** the user can enter more than one airport or region. The query can be constructed, therefore, with two or more origins and destinations appearing in each slice of a journey and the pricing solution would be determined for the various combinations.

Referring now to FIG. **22**, a window **370** is illustrated. Window **370** appears after a user query input to the server process **18** producing the pricing graph **381** that is sent to the client process **36** (FIG. 3). Window **370** is an example of a single slice journey. Since window **370** depicts a one-way (i.e., single slice journey), only controls **352** corresponding to the first slice are shown as activated with information about the slice. Window **370** includes the same controls or buttons **352**, as described above with respect to window **350**. The window **370** also includes a series of user preference controls **354**, here "Nonstop", "Direct", "Online (on the same airline)" and "Select" shown as not activated and "1st class", "2nd class" and "Refundable" shown activated. The Nonstop, Direct and Online controls when selected by a user will eliminate all components from the pricing solution that do not correspond to nonstop, direct or online flights. A select control operates in conjunction with the user marking one or more potential pricing solutions such that the numbers which appear shaded out are activated. When one or more of the pricing solutions are activated and the select button is pressed, the client process extracts pricing solutions from the pricing graph. The "1st class", "2nd class" and "Refundable" controls when activated eliminate fares that do not correspond to these classes of travel.

The window **370** also includes a listing **372** of airports involved in the results provided from the pricing graph **381**, as well as, a listing **374** of airlines.

The window **370** has a graphical region that provides a visual representation of pricing solutions extracted from the pricing graph **38'**. One preferred representation of the pricing solution is a horizontal bar graph **376**. The itineraries are ordered by increasing total fare with each entry **376*a*** of the bar graph corresponding to a set of flight segments on airlines that provide travel from the origin (e.g., 'ESB') to the destination (e.g., SAN, San Diego International Airport) on airlines coded in accordance with the legends for the airline in listing **374** with stopovers denoted by airports. The bar graph representation displays a metric of the pricing solution in a graph format. Thus, as shown for the first entry **376*a***, there are two legs **377*a***, **377*b*** on airline "TK" with a stopover **377*c*** at airport "JFK" and two legs **377*d*** and **377*e*** on airline "HP" arriving in San Diego (SAN). As shown in FIG. **22**, twenty-one possible solutions are represented in the horizontal bar graph ordered by increasing total fare. This typically represents a small fraction of the total number of pricing solutions that may be represented by the pricing graph **38'**. Other ones, if they exist, can be revealed by manipulation of a scroll bar **355**.

Referring now to FIG. **23**, a window **380** illustrates a sample pricing solution including an itinerary **382** and fares **384**. In this embodiment, such a window is produced by double-clicking on an itinerary such as **376*a***. Such a pricing solution is extracted from the pricing-graph **38'** by invalidating **314** all other itineraries in the same slice and then using procedure **304*a*** to find the single cheapest remaining pricing-solution. The window **380** illustrates the sample outbound itinerary **382** with fares **384**. The itinerary **382** can be selected, for example, by double clicking on the itinerary or by using the right button of a computer mouse and so forth. For example, what is displayed in this interface are the itineraries (which are TERMINAL NODES in the pricing graph **38'**) along with their associated "lowest prices" that are computed by running algorithm **304*b*** (with value function such that it computes for every node in the graph the lowest total price for any pricing solution involving the node.

Referring now to FIG. **24**, a second example of a window **390** containing pricing solutions is shown. Window **390** illustrates a round trip journey between Boston (BOS) and San Diego (SAN). The window **390** includes a section including user controls **352** that permit user entry and modification of the results in the window as described above in conjunction with FIG. **22**. Here, however, controls are shown activated for both slice 1 and slice 2 of the journey.

The window also includes the airport **372** and airline lists **374** and a graphical representation **400** of information (e.g., itineraries) with a subsection **402** corresponding to outbound or first slice information (e.g., itineraries) and section **404** corresponding to inbound or second slice information. Each graphical section **402**, **404** is a horizontal bar graph and includes scroll bar controls **403** and **405**, respectively. In addition, the window also includes user preference controls **354** shown activated for the first and the second slices of the journey.

In a similar manner as discussed in conjunction with FIG. **23**, double clicking or otherwise selecting an illustrative ITINERARY, for example, the first ITINERARY **392*a*** will produce an itinerary window **410** (FIG. **25**) that depicts information corresponding to the selected outbound itinerary **412** as well as information for possible return itineraries **414** selected in accordance with the current outbound itinerary. The window **410** also includes fare information **416**.

Referring now to FIG. **26**, a window **390'** is shown. This window depicts a subset of the set of pricing solutions represented by the pricing graph as displayed in window **390** (FIG. **24**). The subset is selected by activating the "Select" button and highlighting a pricing solution (e.g., **392*g***, FIG. **24**). The window **390'** depicts possible return itineraries that can be matched with the selected outbound itinerary **392*g*** and the corresponding total fares for the itinerary. This operation modifies the pricing solutions and is performed within the client process **36**. The client process uses the node invalidating function described in conjunction with FIG. **19**.

Another process that can use the node invalidated function **314** described in conjunction with FIG. **19** would permit the user to point and click at certain airports and/or airlines represented as icons in fields (not numbered). In one mode, by selecting airline and/or airport pricing solutions that do not include the selected airline or airports are not extracted from the pricing graph **38'**. In an alternate mode, selecting an airline or airport does not extract solutions containing the selected airline and/or airport from the pricing graph.

The pricing graph can be viewed in other ways through the activation of the pull down menus shown in any of FIGS. **22**, **24** and **26**. For example, as shown in FIG. **22**, there are four pull down menus "refresh", "outbound display", "return display" and "search properties." The refresh display will permit storing queries and permit a user to refresh the display. The outbound display menu will permit the user to resort the data according to any one of a number of characteristics. Example characteristics include, but are not necessarily limited to, cost, duration, departure time, arrival time, number of legs, number of segments and so forth. The outbound display can also be adjusted to allow a user to

change the horizontal axis to a time or duration axis as well as change the itinerary size to have it displayed small or large. A similar arrangement is provided for the return display. The search properties allow a user to conduct a faring search by computing fares or not computing fares, that is, by providing complete pricing solutions or merely just activating the schedule portion of the server process **18**. The search properties also permit the user to search by legs or segments, specify a search time, a number of itineraries and number of extra itineraries to discard.

Each of the display options referred to above make use of one or more of the value functions and processes described in conjunction with FIG. **19** and permitting the client process to extract or manipulate the pricing graph **38'**. Accordingly, the pull down menus as well as the other controls on the graphical user interface are the user interface to the "value" functions and enumeration processes described above.

Referring now to FIG. **27**, a window **500** is shown. The window **500** includes an ensemble of travel options depicted as a histogram **502**. The histogram **502** is a plot of an metric or option such as time as the x axis versus the number of itineraries on the y axis. Portions of the histogram representation can be selected and the processes above will invalidate all travel node that are not selected. These will provide corresponding changes in a bar graph representation **504** disposed below the histogram **502**. This will also affect a list airports **506** by possible changing a visual appearance of icons in the list **506**.

OTHER EMBODIMENTS

It is to be understood that while the invention has been described in conjunction with the detailed description thereof, the foregoing description is intended to illustrate and not limit the scope of the invention, which is defined by the scope of the appended claims. Other aspects, advantages, and modifications are within the scope of the following claims.

What is claimed is:

1. A travel planning system comprising:

a server process that determines travel planning information in response to a travel request query;

a client process, to send the travel request query and that receives the travel planning information, said client process comprising:

a manipulation process that operates on the travel planning information, to extract, in response to a subsequent user command, travel options from the travel planning information sent from the server process.

2. The system of claim **1** wherein the server process and the client process execute on the same computer.

3. The system of claim **1** wherein said manipulation process of said client process operates on the travel planning information in accordance with at least one user command, and further comprises

a process that produces a set of travel planning information by fare in accordance with the at least one user command.

4. The system of claim **3** wherein said process client further comprises:

a process that sorts said travel planning information by lowest price based on the user input command.

5. The system of claim **1** wherein said manipulation process further comprises:

a process that finds for the set of pricing solutions a pricing solution that optimizes a value function.

6. The system of claim **5** wherein said process that finds a pricing solution, finds for a travel option a best travel option involving an itinerary.

7. The system of claim **1** wherein the manipulation process further comprises:

an enumerating process that uses the set of pricing solutions to produce a subset of the set of pricing solutions in accordance with a user specified preference.

8. The system of claim **7** wherein said enumeration process further comprises:

a process that prunes from the set of pricing solutions, a set of pricing solutions that do not correspond to a user preference.

9. The system of claim **1** further comprising:

a server computer for executing said server process; and

a client computer for executing said client process.

10. The system of claim **9** further comprising a network and wherein said server computer and client computer are interconnected by the network.

11. An airline travel planning system, comprising:

a scheduler that produces a set of itineraries in response to a user specified query;

a faring system that provides fares for the sets of itineraries, and represents the sets of itineraries and fares as a set of logically manipulatable nodes in a data structure where such nodes are shared across itineraries;

an enumeration process that processes the data structure to extract flight-fare components from nodes in the data structure.

12. The system of claim **11** wherein said enumeration process further comprises:

a process that sorts the set of pricing solutions by price in accordance with at least one user preference.

13. The system of claim **11** wherein said process further comprises:

a process that sorts said set of pricing solutions by lowest price.

14. The system of claim **11** wherein the enumeration process further comprises:

a node invalidation process that invalidates nodes in the data structure to produce a subset of valid fares linked to sets of itineraries in accordance with a user specified preference.

15. The system of claim **11** wherein said enumeration process further comprises:

a process that finds for the set of pricing solutions a pricing solution that optimizes a value function.

16. The system of claim **15** wherein said process finds for a travel option a best travel option involving an itinerary.

17. A travel planning system comprising:

a server process that in response to at least one travel destination and at least one travel origin determines a set of pricing solutions, said set of pricing solutions represented by a structure that contains a first plurality of logically combinable entries that represent a second, larger number of pricing solutions;

a client process that receives said pricing solution structure, said client process further comprising:

an enumeration process that extracts pricing solutions from the structure.

18. The system of claim **17** wherein client process comprises:

a query process that causes the enumeration process to extract pricing solutions from the pricing solution structure.

19. The system of claim 17 wherein said pricing solution structure has a number of logical entries which when logically combined can provide a plurality of pricing solutions, with the number of set of pricing solutions being larger than the number of logical entries in the pricing solution structure.

20. The system of claim 17 wherein said client process further comprises:

a process that produces a sorted set of set of pricing solutions by fare in accordance with at least one user preference.

21. The system of claim 17 wherein said client process further comprises:

a process that finds for the set of pricing solutions a pricing solution that optimizes a value function.

22. The system of claim 17 wherein the client process further comprises:

a process that enumerates the set of pricing solutions in accordance with a user preference.

23. The system of claim 17 wherein said process further comprises:

a process that produces a sorted set of said set of pricing solutions by lowest price.

24. The system of claim 23 wherein said process finds for a travel option a best travel option involving an itinerary.

25. The system of claim 24 wherein said enumeration process further comprises:

a process that invalidates logical entries in the pricing solution structure.

26. A travel planning system comprising:

a server process that in response to at least one travel destination and at least one travel origin determines and represents a set of pricing solutions by a directed acyclic graph data structure;

a client process that receives said directed acyclic graph, said client process further comprising:

a process that uses the directed acyclic graph representation to enumerate in response to user preferences a set of pricing solutions.

27. The system of claim 26 wherein said client process further comprises:

a process that produces a sorted set of pricing solutions by price in accordance with at least one user preference.

28. The system of claim 27 wherein said process that produced the sorted set further comprises:

a process that produces the sorted set of said pricing solutions by lowest price.

29. The system of claim 26 wherein said client process further comprises:

an extraction process that extracts set of pricing solutions from the pricing solution structure.

30. The system of claim 29 wherein said client process further comprises:

an invalidate process that invalidates nodes in the directed acyclic graph which are involved only in set of pricing solutions that are outside of a subset of set of pricing solutions.

31. The system of claim 29 wherein said client process further comprises:

a process that finds for the set of pricing solutions a pricing solution that optimizes a value function.

32. The system of claim 31 wherein said process that finds a pricing solution, finds for a travel option a best travel option involving an itinerary.

33. An airline travel planning system comprising:

a server computer;

a server process executed on said server computer, said server process comprising a search process to search for set of pricing solutions in accordance with at least one destination and at least one origin, said search process representing said set of pricing solutions in the form of a directed acyclic graph;

a client computer;

a client computer process responsive to the set of pricing solutions represented in the form of the directed acyclic graph;

said client process further comprising:

a manipulation process that manipulates the set of pricing solutions in the form of the directed acyclic graph representation in response to user preferences, said manipulation process comprising:

a process, responsive to user preferences and to set of pricing solutions represented in the directed acyclic graph, to find a set of pricing solutions in accordance with said user preference;

a process that finds the set of pricing solutions and to produce a subset of said set of pricing solutions in accordance with user specified preferences; and

a sorting process that prunes from the directed acyclic graph nodes that are no longer contained within the subset of set of pricing solutions.

* * * * *